



Obsolete IDL Features



IDL Version 5.6
October, 2002 Edition
Copyright © Research Systems, Inc.
All Rights Reserved

Restricted Rights Notice

The IDL[®], ION Script[™], and ION Java[™] software programs and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. Research Systems, Inc., reserves the right to make changes to this document at any time and without notice.

Limitation of Warranty

Research Systems, Inc. makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

Research Systems, Inc. shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the IDL or ION software packages or their documentation.

Permission to Reproduce this Manual

If you are a licensed user of this product, Research Systems, Inc. grants you a limited, nontransferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

Acknowledgments

IDL[®] is a registered trademark and ION[™], ION Script[™], ION Java[™], are trademarks of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein.

Numerical Recipes[™] is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2[™] is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities
Copyright 1988-2001 The Board of Trustees of the University of Illinois
All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998, 1999, 2000, 2001, 2002 by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library
Copyright © 1999
National Space Science Data Center
NASA/Goddard Space Flight Center

NetCDF Library
Copyright © 1993-1996 University Corporation for Atmospheric Research/Unidata

HDF EOS Library
Copyright © 1996 Hughes and Applied Research Corporation

This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyrighted by INTERSOLV, Inc., 1991-1998.

Use of this software for providing LZW capability for any purpose is not authorized unless user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. For information concerning licensing, please contact: Unisys Corporation, Welch Licensing Department - C1SW19, Township Line & Union Meeting Roads, P.O. Box 500, Blue Bell, PA 19424.

Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Contents

Chapter 1:	
Overview	7
Backwards Compatibility	8
IDL Internal Routines	8
Routines Written in IDL	8
Detecting Use of Obsolete Features	9
Documentation for Older Obsolete Routines	10
 Chapter 2:	
Obsolete Routines	11
DDE Routines	12
DELETE_SYMBOL	13
DELLOG	14
DEMO_MODE	15
DO_APPLE_SCRIPT	16
ERRORF	18
GETHELP	19

GET_SYMBOL	21
HANDLE_CREATE	22
HANDLE_FREE	25
HANDLE_INFO	26
HANDLE_MOVE	28
HANDLE_VALUE	30
HDF_DFSD_ADDDATA	32
HDF_DFSD_DIMGET	34
HDF_DFSD_DIMSET	35
HDF_DFSD_ENDSLICE	37
HDF_DFSD_GETDATA	38
HDF_DFSD_GETINFO	39
HDF_DFSD_GETSLICE	41
HDF_DFSD_PUTSLICE	43
HDF_DFSD_READREF	44
HDF_DFSD_SETINFO	45
HDF_DFSD_STARTSLICE	49
HDF_VD_GETNEXT	51
INP, INPW, OUTP, OUTPW	52
LJLCT	53
PICKFILE	54
POLYFITW	55
REWIND	57
RIEMANN	58
RSTRPOS	63
SET_SYMBOL	65
SETLOG	66
SIZE Executive Command	68
SKIPF	70
SLICER	71
STR_SEP	77
TAPRD	79
TAPWRT	80
TIFF_DUMP	81
TIFF_READ	82
TIFF_WRITE	84

TRNLOG	86
VAX_FLOAT	88
WEOF	90
WIDED	91
WIDGET_MESSAGE	92

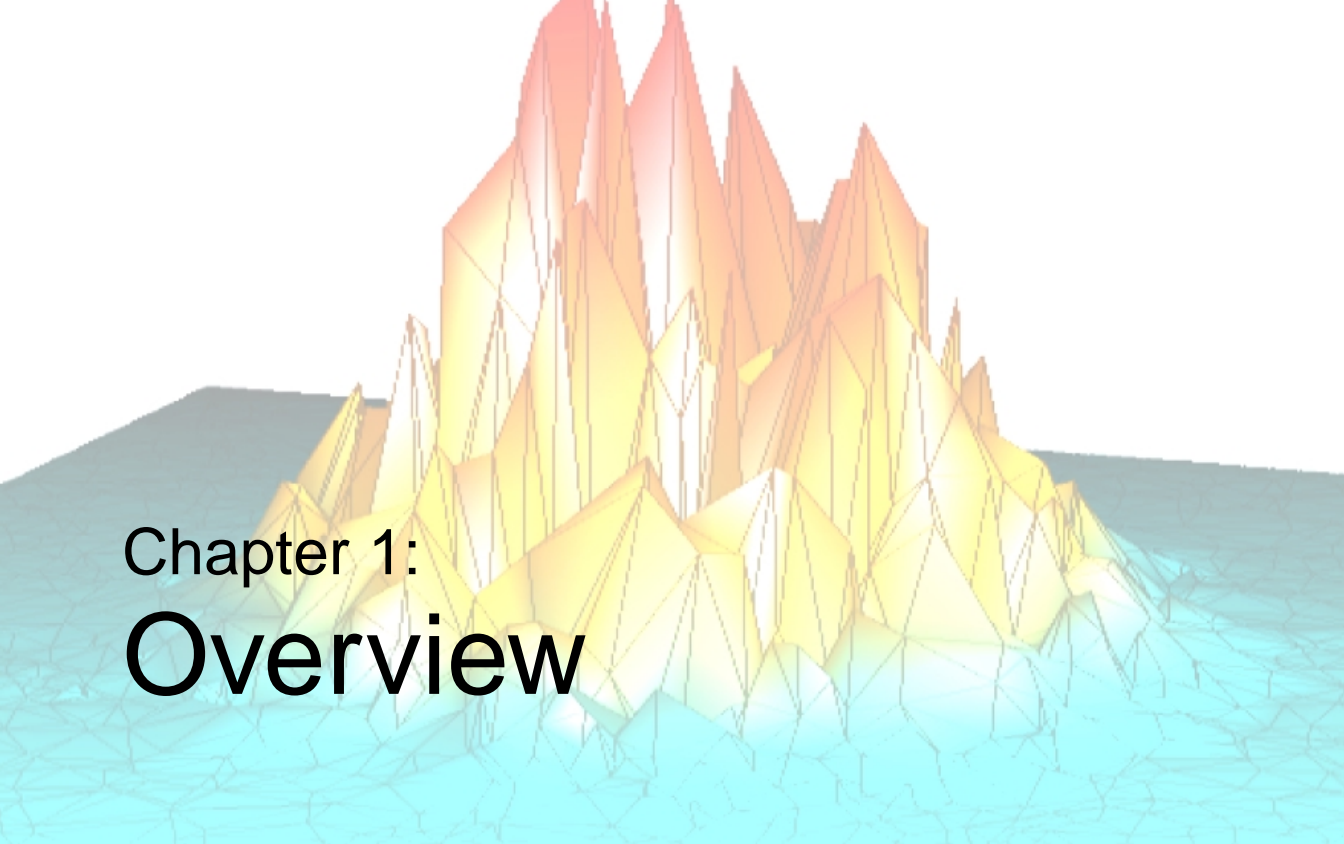
Chapter 3: **Routines with Obsolete Keywords 93**

BYTEORDER	95
CALL_EXTERNAL	96
DEVICE	97
DOC_LIBRARY	98
EXTRACT_SLICE	99
IDLgrMPEG::Save	100
IDLgrVolume::Init	101
LINKIMAGE	102
LIVE_PRINT	103
MAKE_DLL	104
ONLINE_HELP	105
OPEN	106
PRINT/PRINTF	112
READ_TIFF	113
READ/READF	114
READU	115
SAVE	116
SPAWN	117
WIDGET_BASE	119
WRITE_TIFF	120
WRITEU	121

Chapter 4: **Obsolete Graphics Devices 123**

The LJ Device	124
LJ Driver Strengths	125
LJ Driver Limitations	125
LJ Suggestions	126
The Macintosh Device	127

Chapter 5:	
Remote Procedure Calls	129
Using IDL as an RPC Server	131
The IDL RPC Directory	131
Running IDL in Server Mode	131
Creating the IDL RPC Library	131
Linking your Client Program	132
The IDL RPC Library	133
free_idl_variable	134
get_idl_variable	135
idl_server_interactive	137
kill_server	138
register_idl_client	139
send_idl_command	140
set_idl_timeout	141
set_idl_variable	142
set_rpc_verbosity	144
unregister_idl_client	145
The varinfo_t Structure	146
Variable Creation Functions	146
v_make_byte	147
v_make_complex	148
v_make_dcomplex	149
v_make_double	150
v_make_float	151
v_make_int	152
v_make_long	153
v_make_string	154
v_fill_array	155
More Variable Manipulation Macros	156
Notes on Variable Creation and Memory Management	158
Freeing Resources	158
Creating a Statically-Allocated Array	158
Allocating Space for Strings	159
RPC Examples	160



Chapter 1: Overview

This chapter discusses the following topics:

Backwards Compatibility	8	Documentation for Older Obsolete Routines .	10
Detecting Use of Obsolete Features	9		

Backwards Compatibility

Research Systems strongly recommends that you not use obsolete routines when writing new IDL code. As IDL continues to evolve, the likelihood that obsolete routines will no longer function as expected increases. While we will continue to make every effort to ensure that obsolete routines shipped with IDL function, in a small number of cases this may not be possible.

IDL Internal Routines

Routines that are built into the IDL executable—routines *not* written in the IDL language—will either continue to be included in the executable until the scheduled removal release or will be re-implemented in the IDL language. In the latter case, obsolete routines may run slower than the original version. Note that obsolete routines that have been re-implemented in the IDL language may also be scheduled for eventual removal.

Routines Written in IDL

Routines written in the IDL language (`.pro` files) are contained in the obsolete subdirectory of the `lib` directory of the IDL distribution. As long as a given obsolete routine is included in this subdirectory, it will continue to function as always.

Detecting Use of Obsolete Features

You can search for usage of obsolete routines, system variables, and syntax by setting the fields of the `!WARN` system variable. Setting `!WARN` causes IDL to print informational messages to the command log or console window when it encounters references to obsolete features. See “[!WARN](#)” in the *IDL Reference Guide* manual for details.

Documentation for Older Obsolete Routines

Routines that became obsolete in IDL version 4.0 or earlier are not documented in this book or in the IDL Online Help. However, if the routine is written in the IDL language, you can inspect the documentation header of the `.pro` file, or use the [DOC_LIBRARY](#) routine. The `.pro` files for obsolete routines are located in the `obsolete` subdirectory of the `lib` directory of the IDL distribution.



Chapter 2: Obsolete Routines

This chapter contains complete documentation for obsoleted IDL routines. New IDL code should not use these routines. For a list of the routines that replace each of these obsolete routines, see [Appendix O, “Obsolete Features”](#) in the *IDL Reference Guide* manual.

DDE Routines

These routines are obsolete and should not be used in new IDL code.

Windows-Only Routines for Dynamic Data Exchange (DDE)

IDL for Windows supports DDE client capability for cold DDE links. The relevant system calls are documented below:

Result = DDE_GETSERVERS()

This function returns an array of service names for the currently-available DDE servers.

Result = DDE_GETTOPICS(*server*)

This function returns the topics list for the specified server. The server argument is a scalar string containing the name of the desired DDE server.

Result = DDE_GETITEMS(*server*)

This function returns the items list for the specified server. The server argument is a scalar string containing the name of the desired DDE server.

Result = DDE_REQUEST(*server, topic, item*)

This function returns the requested data in string format. The server, topic, and item arguments must be scalar strings.

DDE_EXECUTE, *server, topic, command*

This procedure causes the DDE server to execute the command for the specified topic. The server, topic, and command arguments must be scalar strings.

DELETE_SYMBOL

The DELETE_SYMBOL procedure deletes a DCL (Digital Command Language) interpreter symbol for the current process.

Note

This procedure is available on VMS only.

Syntax

```
DELETE_SYMBOL, Name [, TYPE={1 | 2}]
```

Arguments

Name

A scalar string containing the name of the symbol to be deleted.

Keywords

TYPE

Indicates the table from which *Name* will be deleted. Set TYPE to 1 to specify the local symbol table. Set TYPE to 2 to specify the global symbol table. The default is to search the local table.

DELLOG

The DELLOG procedure deletes a VMS logical name.

Note

This procedure is available on VMS only.

Syntax

DELLOG, *Lognam* [, TABLE=*string*]

Arguments

Lognam

A scalar string containing the name of the logical to be deleted.

Keywords

TABLE

A scalar string giving the name of the logical table from which to delete *Lognam*. If TABLE is not specified, LNM\$PROCESS_TABLE is used.

DEMO_MODE

This routine is obsolete and should not be used in new IDL code.

The DEMO_MODE function returns True if IDL is running in the timed demo mode (i.e., a license manager is not running). Calling this function causes a FLUSH, -1 command to be issued.

Syntax

Result = DEMO_MODE()

DO_APPLE_SCRIPT

This routine is obsolete and should not be used in new IDL code.

The DO_APPLE_SCRIPT procedure compiles and executes an AppleScript script, possibly returning a result. DO_APPLE_SCRIPT is only available in IDL for Macintosh.

Syntax

DO_APPLE_SCRIPT, *Script* [, /AG_STRING] [, RESULT=*variable*]

Arguments

Script

A string or array of strings to be compiled and executed by AppleScript.

Keywords

AS_STRING

Set this keyword to cause the result to be returned as a decompiled string. Decompiled strings have the same format as the “The Result” window of Apple’s Script Editor.

RESULT

Set this keyword equal to a named variable that will contain the results of the script.

Example

Suppose you wish to retrieve a range of cell data from a Microsoft Excel spreadsheet. The following AppleScript script and command retrieve the first through fifth rows of the first two columns of a spreadsheet titled “Worksheet 1”, storing the result in the IDL variable A:

```
script = [ 'tell application "Microsoft Excel"', $
  'get Value of Range "R1C1:R5C2" of Worksheet 1', $
  'end tell' ]
DO_APPLE_SCRIPT, script, RESULT = a
```

Similarly, the following lines would copy the contents of the IDL variable A to a range within the spreadsheet:


```
A = [ 1, 2, 3, 4, 5 ]
script = [ 'tell application "IDL" to copy variable "A"', $
          'into aVariable', $
          'tell application "Excel" to copy aVariable to', $
          'value of range "R1C1:R5C1" of worksheet 1' ]
DO_APPLE_SCRIPT, script
```

ERRORF

This routine is obsolete and should not be used in new IDL code.

The ERRORF function returns the value of the error function:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

The result is double-precision if the argument is double-precision. If the argument is floating-point, the result is floating-point. The result always has the same structure as *X*. The ERRORF function does not work with complex arguments.

Syntax

Result = ERRORF(*X*)

Arguments

X

The expression for which the error function is to be evaluated.

Example

To find the error function of 0.4 and print the result, enter:

```
PRINT, ERRORF(0.4)
```

IDL prints:

```
0.428392
```

GETHELP

This routine is obsolete and should not be used in new IDL code.

The GETHELP function returns information on variables defined at the program level from which GETHELP is called. The function builds a string array that contains information that follows the format used by the IDL HELP command.

When called without an argument, GETHELP returns a string array that normally contains variable data that is in the same format as used by the IDL HELP procedure. The variables in this list are those defined for the routine (or program level) that called GETHELP. If there are no variables defined, or the specified variable does not exist, GETHELP returns a null string. Other information can be obtained by setting keywords.

Syntax

Result = GETHELP(*[Variable]*)

Arguments

Variable

A scalar string that contains the name of the variable from which to get information. If this argument is omitted, GETHELP returns an array of strings where each element contains information on a separate variable, one element for each defined variable.

Keywords

FULLSTRING

Normally a string that is longer than 45 chars is truncated and followed by “...” just like the HELP command. Setting this keyword will cause the full string to be returned.

FUNCTIONS

Setting this keyword will cause the function to return all current IDL compiled functions.

ONELINE

If a variable name is greater than 15 characters it is usually returned as 2 two elements of the output array (Variable name in 1st element, variable info in the 2nd element). Setting this keyword will put all the information in one string, separating the name and data with a space.

PROCEDURES

Setting this keyword will cause the function to return all current IDL compiled procedures.

SYS_PROCS

Setting this keyword will cause the function to return the names of all IDL system (built-in) procedures.

SYS_FUNCS

Setting this keyword will cause the function to return the names of all IDL system (built-in) functions.

Note

RESTRICTIONS: Due to the difficulties in determining if a variable is of type associate, the following conditions will result in the variable being listed as a structure. These conditions are:

- Associate record type is structure.
- Associated file is opened for update (openu).
- Associate file is not empty.

Another difference between this routine and the IDL help command is that if a variable is in a common block, the common block name is not listed next to the variable name. Currently there is no method available to get the common block names used in a routine.

Example

To obtain a listing in a help format of the variables contained in the current routine you would make the following call:

```
HelpData = GetHelp()
```

The variable HelpData would be a string array containing the requested information.

GET_SYMBOL

This routine is obsolete and should not be used in new IDL code.

The GET_SYMBOL function returns the value of a VMS DCL (Digital Command Language) interpreter symbol as a scalar string. If the symbol is undefined, the null string is returned.

Note

This procedure is available on VMS only.

Syntax

Result = GET_SYMBOL(*Name* [, TYPE={1 | 2}])

Arguments

Name

A scalar string containing the name of the symbol to be translated.

Keywords

TYPE

The table from which *Name* is translated. Set TYPE to 1 to specify the local symbol table. A value of 2 specifies the global symbol table. The default is to search the local table.

HANDLE_CREATE

This routine is obsolete and should not be used in new IDL code.

The `HANDLE_CREATE` function creates a new handle. A “handle” is a dynamically-allocated variable that is identified by a unique integer value known as a “handle ID”. Handles can have a value, of any IDL data type and organization, associated with them. This function returns the handle ID of the newly-created handle.

Because handles are dynamic, they can be used to create complex data structures. They are also global in scope, but do not suffer from the limitations of `COMMON` blocks. That is, handles are available to all program units at all times. (Remember, however, that IDL variables containing handle IDs are not global in scope and must be declared in a `COMMON` block if you want to share them between program units.)

Handle Terminology

The following terms are used to describe handles in the documentation for this function and other handle-related routines:

- **Handle ID:** The unique integer identifier associated with a handle.
- **Handle value:** Data of any IDL type and organization associated with a handle.
- **Top-level handle:** A handle at the top of a handle hierarchy. A top-level handle can have children, but does not have a parent.
- **Parents, children, and siblings:** These terms describe the relationship between handles in a handle hierarchy. When a new handle is created, it can be the start of a new handle hierarchy (a top-level handle) or it can belong to the level of a handle hierarchy below an existing handle. A handle created in this way is said to be a child of the specified parent. Parents can have any number of children. All handles that share the same parent are said to be siblings.

Syntax

Result = `HANDLE_CREATE`(*ID*)

Arguments

ID

If this argument is present, it specifies the handle ID relative to which the new handle is created. Normally, the new handle becomes the last child of the parent handle specified by ID. However, this behavior can be changed by setting the `FIRST_CHILD` or `SIBLING` keywords.

Omit this argument to create a new top-level handle without a parent.

Keywords

FIRST_CHILD

Set this keyword to create the new handle as the first child of the handle specified by ID. Any existing children of ID become later siblings of the new first child (i.e., the existing first child becomes the second child, the second child becomes the third child, etc.).

NO_COPY

Usually, when the `VALUE` keyword is used, the source variable memory is copied to the handle value. If the `NO_COPY` keyword is set, the value data is taken away from the source variable and attached directly to the destination. This feature can be used to move data very efficiently. However, it has the side effect of causing the source variable to become undefined.

SIBLING

Set this keyword to create the new handle as the sibling handle immediately following ID. Any other siblings currently following ID become later siblings of the new handle. Note that you cannot create a handle that is a sibling of a top-level handle.

VALUE

The value to be assigned to the handle.

Every handle can contain a user-specified value of any data type and organization. This value is not used by the handle in any way, but exists entirely for the convenience of the IDL programmer. Use this keyword to set the handle value when the handle is first created.

If the `VALUE` keyword is not specified, the handle's initial value is undefined.

Handle values can be retrieved using the `HANDLE_VALUE` procedure.

Examples

The following commands create a top-level handle with 3 child handles. Each handle is assigned a different string value:

```
;Create top-level handle without an initial handle value:
top = HANDLE_CREATE()
;Create first child of the top-level handle:
first = HANDLE_CREATE(top, VALUE='First child')
;Create second child of the top-level handle:
second = HANDLE_CREATE(top, VALUE='Second child')
;Create a new sibling between first and second.
;This handle is also a child of the top-level handle:
third = HANDLE_CREATE(first, VALUE='Another child', /SIBLING)
```


HANDLE_FREE

This routine is obsolete and should not be used in new IDL code.

The `HANDLE_FREE` procedure frees an existing handle, along with any dynamic memory currently being used by its value. Any child handles associated with `ID` are also freed.

Syntax

```
HANDLE_FREE, ID
```

Arguments

ID

The ID of the handle to be freed. Once the handle is freed, further use of it is invalid and causes an error to be issued.

Example

To free all memory associated with the top-level handle `top`, and all its children, use the command:

```
HANDLE_FREE, top
```

HANDLE_INFO

This routine is obsolete and should not be used in new IDL code.

The `HANDLE_INFO` function returns information about handle validity and connectivity. By default, it returns `True` if the specified handle ID is valid. Keywords can be set to return other types of information.

Syntax

```
Result = HANDLE_INFO(ID)
```

Arguments

ID

The ID of the handle for which information is desired. This argument can be scalar or array an array of IDs. The result of `HANDLE_INFO` has the same structure as `ID`, and each element gives the desired information for the corresponding element of `ID`.

Keywords

FIRST_CHILD

Set this keyword to return the handle ID of the first child of the specified handle. If the handle has no children, 0 is returned.

NUM_CHILDREN

Set this keyword to return the number of children related to ID.

PARENT

Set this keyword to return the handle ID of the parent of the specified handle. If the specified handle is a top-level handle (i.e., it has no parent), 0 is returned.

SIBLING

Set this keyword to return the handle ID of the sibling handle following ID. If ID has no later siblings, or if ID is a top-level handle, 0 is returned.

VALID_ID

Set this keyword to return 1 if ID represents a currently valid handle. Otherwise, zero is returned. This is the default action for `HANDLE_INFO` if no other keywords are specified.

Examples

The following commands demonstrate a number of different uses of `HANDLE_INFO`:

```
;Print a message if handle1 is a valid handle ID.  
IF HANDLE_INFO(handle1) THEN PRINT, 'Valid handle.'  
;Retrieve the handle ID of the first child of top.  
handle = HANDLE_INFO(top, /FIRST_CHILD)  
;Retrieve the handle ID of the next sibling of handle1.  
next= HANDLE_INFO(handle1, /SIBLING)
```

HANDLE_MOVE

This routine is obsolete and should not be used in new IDL code.

The `HANDLE_MOVE` procedure moves a handle (specified by `Move_ID`) to a new location. This new position is specified relative to `Static_ID`.

Syntax

`HANDLE_MOVE, Static_ID, Move_ID`

Arguments

Static_ID

The handle ID relative to which the handle specified by `Move_ID` is moved. By default, `Move_ID` becomes the last child of `Static_ID`. This behavior can be changed by specifying one of the keywords described below.

If `Static_ID` is set to 0, `Move_ID` becomes a top level handle without any parent. `Static_ID` cannot be a child of `Move_ID`.

Move_ID

The ID of the handle to be moved.

Keywords

FIRST_CHILD

Set this keyword to make `Move_ID` the first child of `Static_ID`. Any existing children of `Static_ID` become later siblings of the new first child (i.e., the existing first child becomes the second child, the second child becomes the third child, etc.).

SIBLING

Set this keyword to make `Move_ID` the sibling handle immediately following `Static_ID`. Any siblings currently following `Static_ID` become later siblings of the new handle. Note that you cannot move a handle such that it becomes a sibling of a top-level handle.

Example

```
; Create top-level handle:
top = HANDLE_CREATE()
; Create first child of top:
child1 = HANDLE_CREATE(top)
; Create second child of top:
child2 = HANDLE_CREATE(top)
; Move the first child to be the last child of top:
HANDLE_MOVE, top, child1
```

HANDLE_VALUE

This routine is obsolete and should not be used in new IDL code.

The HANDLE_VALUE procedure returns or sets the value of an existing handle.

Syntax

HANDLE_VALUE, *ID*, *Value*

Arguments

ID

A valid handle ID.

Value

When using HANDLE_VALUE to return an existing handle value (the default), Value is a named variable in which the value is returned.

When using HANDLE_VALUE to set a handle value, Value is the new value. Note that handle values can have any IDL data type and organization.

Keywords

NO_COPY

By default, HANDLE_VALUE works by making a second copy of the source data. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the NO_COPY keyword is set, HANDLE_VALUE works differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used to move data very efficiently. However, it has the side effect of causing the source variable to become undefined. On a retrieve operation, the handle value becomes undefined. On a set operation, the variable passed as Value becomes undefined.

SET

Set this keyword to assign Value as the new handle value. The default is to retrieve the current handle value.

Example

The following commands demonstrate the two different uses of `HANDLE_VALUE`:

```
; Retrieve the value of handle1 into the variable current:
HANDLE_VALUE, handle1, current
; Set the value of handle1 to a 2-element integer vector:
HANDLE_VALUE,handle1,[2,3],/SET
```

HDF_DFSD_ADDDATA

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_ADDDATA procedure writes data, as well as all other information set via calls to HDF_DFSD_SETINFO and HDF_DFSD_DIMSET, to an HDF file.

The *Data* array must have the same dimensions as the array in the file. The new SDS is appended to the file, unless the OVERWRITE keyword is set.

Syntax

```
HDF_DFSD_ADDDATA, Filename, Data [, /OVERWRITE]  
[, SET_DIM=value{must set either this or the DIMS keyword to  
HDF_DFSD_SETINFO}] [, /SET_TYPE]
```

Arguments

Filename

A scalar string containing the name of the file to be written.

Data

An expression (typically an array) containing the data to write.

Keywords

OVERWRITE

Set this keyword to write *Data* as the first, and only, SDS in the file. All previously-written scientific data sets in the file are removed.

SET_DIM

Set this keyword to make the dimension information for the HDF file based upon the dimensions of *Data*.

Note

You *must* set the number of dimensions in the HDF file, either by setting the SET_DIM keyword or using the DIMS keyword to HDF_DFSD_SETINFO.

SET_TYPE

Set this keyword to make the data type of the current SDS based on the data type of the Data argument.

HDF_DFSD_DIMGET

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_DIMGET procedure retrieves information about the specified dimension number of the current HDF file.

Syntax

```
HDF_DFSD_DIMGET, Dimension [, /FORMAT] [, /LABEL] [, SCALE=vector]  
[, /UNIT]
```

Arguments

Dimension

The dimension number [0, 1, 2, ...] to get information about.

Keywords

FORMAT

Set this keyword to return the dimension format string.

LABEL

Set this keyword to return the dimension label string.

SCALE

Use this keyword to return scale information about the dimension. Set this keyword to a vector of values of the same type as the data.

UNIT

Set this keyword to return the dimension unit string.

HDF_DFSD_DIMSET

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_DIMSET procedure sets the label, unit, format, or scale of dimensions in an HDF. Note that the label, unit, and format of a dataset must be set simultaneously.

Syntax

```
HDF_DFSD_DIMSET, Dimension [, FORMAT=string] [, LABEL=string]  
[, SCALE=vector] [, UNIT=string]
```

Arguments

Dimension

The dimension number that the label, unit, format or scale apply to.

Keywords

FORMAT

A string for the dimension format. This string should be a standard IDL formatting string.

LABEL

A string for the dimension label.

SCALE

A vector of values used to set the dimension scale.

UNIT

A string for the dimension units.

Example

Suppose that a stored dataset is a 20 by 100 by 50 element floating-point array of values representing water content within the volume of a cloud. Assume further that each element in the 100-element dimension (the “Y” dimension) was sampled at 1/10

mile increments. Appropriate labeling, formatting, unit, and scaling information for the Y dimension can be set with the following command:

```
HDF_DFSD_DIMSET, 1, LABEL = 'Y Position', FORMAT = 'F8.2', $  
    UNIT = 'Miles', SCALE = 0.1*FINDGEN(100)
```

HDF_DFSD_ENDSLICE

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_ENDSLICE procedure ends a sequence of calls started by HDF_DFSD_STARTSLICE by closing the internal slice interface and synchronizing the file.

Syntax

HDF_DFSD_ENDSLICE

Example

See the example in the documentation for HDF_DFSD_STARTSLICE.

HDF_DFSD_GETDATA

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_GETDATA procedure reads data from an HDF file.

Syntax

```
HDF_DFSD_GETDATA, Filename, Data [, /GET_DIMS{Set only if you have not  
called HDF_DFSD_GETINFO with the DIMS keyword}] [, /GET_TYPE]
```

Arguments

Filename

A scalar string containing the name of the file to be read.

Data

A named variable in which the data is returned.

Keywords

GET_DIMS

Set this keyword to get dimension information for reading the data. This keyword should only be used if one has *not* called HDF_DFSD_GETINFO with the DIMS keyword

GET_TYPE

Set this keyword to get the data type for the current SDS.

HDF_DFSD_GETINFO

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_GETINFO procedure retrieves information about the current HDF file.

Note that calling HDF_DFSD_GETINFO with the DIMS or TYPE keywords may alter which dataset is current. See “Reading an Entire Scientific Dataset” and “Getting Other Information About SDSs” in the *NCSA HDF Calling Interfaces and Utilities* documentation.

Note that reading a label, unit, format, or coordinate system string that has more than 256 characters can have unpredictable results.

Syntax

```
HDF_DFSD_GETINFO, Filename [, CALDATA=variable] [, /COORDSYS]
[, DIMS=variable] [, /FORMAT] [, /LABEL] [, /LASTREF] [, /NSDS] [, /RANGE]
[, TYPE=variable] [, /UNIT]
```

Arguments

Filename

A scalar string containing the name of the file to be read. A filename is only needed to determine SDS dimensions and/or the number of SDSs in a file.

Keywords

CALDATA

Set this keyword to a named variable which will contain the calibration data associated with an SDS data set. The data will be returned in a structure of the form:

```
{ CAL: 0d, CAL_ERR: 0d, OFFSET: 0d, $
  OFFSET_ERR: 0d, NUM_TYPE: 0L }
```

COORDSYS

Set this keyword to return the data coordinate system description string.

DIMS

Set this keyword to a named variable in which the dimensions of the current SDS are returned in a longword array.

FORMAT

Set this keyword to return the data format description string.

LABEL

Set this keyword to return the data label description string.

LASTREF

Set this keyword to return the last reference number written or read for an SDS.

NSDS

Set this keyword to return the number of SDSs in the file.

RANGE

Set this keyword to return the valid max/min values for the current SDS.

TYPE

Set this keyword to a named variable which returns a string describing the type of the current SDS (e.g., 'BYTE', 'FLOAT', etc.).

UNIT

Set this keyword to return the data unit description string.

Example

The following commands read an SDS, including information about its dimensions but not its annotations:

```
HDF_DFSD_GETINFO, filename, DIMS=d, TYPE=t, RANGE=r, $
    LABEL=l, UNIT=u, FORMAT=f, COORDSYS=c
...
FOR i = 0, N_ELEMENTS(d)-1 DO BEGIN
    HDF_DFSD_DIMGET, i, LABEL=dl, UNIT=du, FORMAT=df, SCALE=ds
ENDFOR
HDF_DFSD_GETDATA, filename, data
```


HDF_DFSD_GETSLICE

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_GETSLICE procedure reads a slice of data from the current Hierarchical Data Format file.

Note

Before calling HDF_DFSD_GETSLICE, call HDF_DFSD_GETINFO with the DIMS and TYPE keywords to get the dimensions and type of the next data slice. Failure to get the dimensions and type will cause the HDF interface to attempt to read the data incorrectly, and may cause unexpected results.

Syntax

HDF_DFSD_GETSLICE, *Filename*, *Data* [, COUNT=*vector*] [, OFFSET=*vector*]

Arguments

Filename

A scalar string containing the name of the file to be read.

Data

A named variable in which the data, read from the SDS, is returned.

Keywords

COUNT

An optional vector containing the counts to be used in reading Value. The default is to read all elements in each record taking the value of OFFSET into account.

OFFSET

A vector specifying the array indices within the specified record at which to begin reading. OFFSET is a 1-dimensional array containing one element per HDF dimension. The default value is zero for each dimension.

Example

See the example in the documentation for `HDF_DFSD_STARTSLICE`.

HDF_DFSD_PUTSLICE

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_PUTSLICE procedure writes a data slice to the current HDF file.

Note

Before calling HDF_DFSD_PUTSLICE, call HDF_DFSD_SETINFO to set the dimensions and attributes of the slice and HDF_DFSD_STARTSLICE to initialize the slice interface.

Syntax

HDF_DFSD_PUTSLICE, *Data* [, COUNT=*vector*]

Arguments

Data

An array containing the data to write. Dimensions used to write the data are taken from the dimensions of *Data*, unless the COUNT keyword is used.

Keywords

COUNT

An optional vector containing the counts to be used in writing *Data*. The counts do have to match the dimensions (number or sizes), but the count cannot describe more elements than exist.

Example

See the example in the documentation for HDF_DFSD_STARTSLICE.

HDF_DFSD_READREF

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_READREF procedure specifies the reference number of the HDF file to be read by the next call to HDF_DFSD_GETINFO or HDF_DFSD_GETDATA.

Syntax

HDF_DFSD_READREF, *Filename*, *Refno*

Arguments

Filename

A scalar string containing the name of the file to be read.

Refno

The reference number of the desired SDS.

HDF_DFSD_SETINFO

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_SETINFO procedure controls information associated with an HDF file. Because of the manner in which the underlying HDF library was written, it is necessary to set the dimensions and data type of a scientific data set the first time that HDF_DFSD_SETINFO is called.

This procedure has many options, controlled by keywords. The order in which the keywords are specified is unimportant as the routine insures the order of operation for any given call to it. CLEAR and RESTART requests are performed first, followed by type and dimension setting, followed by length setting, followed by the remaining keyword requests.

If you are not writing any ancillary information, you can call HDF_DFSD_ADDDATA with the SET_TYPE and/or SET_DIMS keywords.

Data string lengths should be set before, or at the same time as, writing the corresponding data string. For example:

```
HDF_DFSD_SETINFO, LEN_FORMAT=10, FORMAT='12.3F'
```

or

```
HDF_DFSD_SETINFO, LEN_FORMAT=10
HDF_DFSD_SETINFO, FORMAT='12.3F'
```

Due to the underlying C routines, it is necessary to set all four data strings at the same time, or the unspecified strings are treated as "" (null strings).

For example:

```
HDF_DFSD_SETINFO, LABEL = 'hi'
HDF_DFSD_SETINFO, UNIT = 'ergs'
```

is the same as:

```
HDF_DFSD_SETINFO, LABEL='hi', UNIT='', FORMAT='', COORDSYS=''
HDF_DFSD_SETINFO, LABEL='', UNIT='ergs', FORMAT='', COORDSYS=''
```

Syntax

```
HDF_DFSD_SETINFO [, CALDATA=structure] [, /CLEAR]
[, COORDSYS=string] [, DIMS=vector] [, /BYTE | /DOUBLE | /FLOAT | /INT |
/, /LONG] [, FORMAT=string] [, LABEL=string] [, LEN_LABEL=value]
[, LEN_UNIT=value] [, LEN_FORMAT=value] [, LEN_COORDSYS=value]
[, RANGE=[max, min]] [, /RESTART] [, UNIT=string]
```

Arguments

None

Keywords

BYTE

Set this keyword to make the SDS data type DFNT_UINT8 (1-byte unsigned integer).

CALDATA

Set this keyword to a structure containing calibration information. The structure should contain five tags, the first four of which are double-precision floating-point, and fifth of which should be long integer. For example:

```
caldata = { Cal:      1.0d $ ; Calibration factor.
            Cal_Err:  0.1d $ ; Calibration error.
            Offset:   2.5d $ ; Uncalibrated offset.
            Offset_Err: 0.1d $ ; Uncalibrated offset error.
            Num_Type:  5L   $ ; Number type of uncalib.data.
```

Some typical values for the Num_Type field include:

For byte data:

```
3L      (DFNT_UCHAR8)
21L     (DFNT_UINT8)
```

For integer data:

```
22L     (DNFT_INT16)
```

For long-integer data:

```
24L     (DFNT_INT32)
```

For floating-point data:

```
5L      (DFNT_FLOAT32)
6L      (DFNT_FLOAT64)
```

There are other types, but they are not native to IDL. They can be found in the `hdf.h` header file for the HDF library.

CLEAR

Set this keyword to reset all possible set values to their default value.

COORDSYS

A string for the data coordinate system description.

DIMS

Set this keyword to a vector of dimensions to be used in writing the next SDS. For example:

```
HDF_DFSD_SETINFO, DIMS = [10, 20, 30]
```

DOUBLE

Set this keyword to make the SDS data type DFNT_FLOAT64 (8-byte floating point).

FLOAT

Set this keyword to make the SDS data type DFNT_FLOAT32 (4-byte floating point).

FORMAT

A string for the data format description.

INT

Set this keyword to make the SDS data type DFNT_INT16 (2-byte signed integer).

LABEL

A string for the data label description.

LEN_LABEL

The label string length (default is 255).

LEN_UNIT

The unit string length (default is 255).

LEN_FORMAT

The format string length (default is 255).

LEN_COORDSYS

The format coordinate system string length (default is 255).

LONG

Set this keyword to make the SDS data type DFNT_INT32 (4-byte signed integer).

RANGE

The minimum and maximum range, represented as a 2-element vector of the same data type as the data to be written. The first element is the maximum, the second is the minimum. For example:

```
HDF_DFSD_SETINFO, RANGE = [10,0]
```

RESTART

Set this keyword to make the get (HDF_DFSD_GETSLICE) routine read from the first SDS in the file.

UNIT

A string for the data unit description.

Example

Write a 100x50 array of longs:

```
data = LONARR(100, 50)
HDF_DFSD_SETINFO, /CLEAR, /LONG, DIMS=[100,50], $
  RANGE=[MAX(data), MIN(data)], $
  LABEL='pressure', UNIT='pascals', $
  FORMAT='F10.0'
```


HDF_DFSD_STARTSLICE

This routine is obsolete and should not be used in new IDL code.

The HDF_DFSD_STARTSLICE procedure prepares the system to write a slice of data to an HDF file. HDF_DFSD_SETINFO must be called before HDF_DFSD_STARTSLICE to set the dimensions and attributes of the slice.

This procedure must be called before calling HDF_DFSD_PUTSLICE, and must be terminated with a call to HDF_DFSD_ENDSLICE.

Syntax

HDF_DFSD_STARTSLICE, *Filename*

Arguments

Filename

A scalar string containing the name of the file to be written.

Example

```
; Open an HDF file:
fid=HDF_OPEN('test.hdf',/ALL)

; Create two datasets:
slicedata1=FOUNDGEN(5,10,15)
slicedata2=FOUNDGEN(4,5)

; Use HDF_DFSD_SETINFO to set the dimensions, then add
; the first slice:
HDF_DFSD_SETINFO,LABEL='label1', DIMS=[5,10,15], /FLOAT
HDF_DFSD_STARTSLICE,'test.hdf'
HDF_DFSD_PUTSLICE, slicedata1
HDF_DFSD_ENDSLICE

; Repeat the process for the second slice:
HDF_DFSD_SETINFO, LABEL='label2', DIMS=[4,5], /DOUBLE
HDF_DFSD_STARTSLICE,'test.hdf'
HDF_DFSD_PUTSLICE, slicedata2
HDF_DFSD_ENDSLICE
HDF_DFSD_SETINFO, /RESTART

; Use HDF_DFSD_GETINFO to advance slices and set slice
```

```
; attributes, then get the slices:
HDF_DFSD_GETINFO, name, DIMS=dims, TYPE=type
HDF_DFSD_GETSLICE, out1
HDF_DFSD_GETINFO, name, DIMS=dims, TYPE=type
HDF_DFSD_GETSLICE, out2

; Close the HDF file:
HDF_CLOSE('test.hdf')

;Check the first slice to see if everything worked:
IF TOTAL(out1 EQ slicedata1) EQ N_ELEMENTS(out1) THEN $
    PRINT, 'SLICE 1 WRITTEN/READ CORRECTLY' ELSE $
    PRINT, 'SLICE 1 WRITTEN/READ INCORRECTLY'
; Check the second slice to see if everything worked:
IF TOTAL(out2 EQ slicedata2) EQ N_ELEMENTS(out2) THEN $
    PRINT, 'SLICE 2 WRITTEN/READ CORRECTLY' ELSE $
    PRINT, 'SLICE 2 WRITTEN/READ INCORRECTLY'
```

IDL Output

```
SLICE 1 WRITTEN/READ CORRECTLY

SLICE 2 WRITTEN/READ CORRECTLY
```

HDF_VD_GETNEXT

The HDF_VD_GETNEXT function returns the reference number of the next object inside a VData in an HDF file. If Id is -1, the first item in the VData is returned, otherwise Id should be set to a reference number previously returned by HDF_VD_GETNEXT. HDF_VD_GETNEXT returns -1 if there was an error or there are no more objects after the one specified by Id.

Syntax

Result = HDF_VD_GETNEXT(*VData*, *Id*)

Arguments

VData

The VData handle returned by a previous call to HDF_VD_ATTACH.

Id

A VGroup or VData reference number obtained by a previous call to HDF_VG_GETNEXT or HDF_VD_GETNEXT. Alternatively, this value can be set to -1 to return the first item in the VData.

Version History

Introduced: 4.0

INP, INPW, OUTP, OUTPW

These routines are obsolete and should not be used in new IDL code.

Windows-Only Routines for Hardware Ports

You can address the hardware ports of your personal computer directly using the following routines. In each case, *Port* is specified using the hexadecimal address of the hardware port. For example, if serial port #1 of your PC is at address 3F8, you would use the following IDL commands to read that port:

```
paddr = '3F8'xSet paddr to hexadecimal value.
data = INPW(paddr)Read data.
```

Result = INP(Port, [$D_1 \dots D_N$])

This function returns either one byte (if only the port number is specified) or an array (the dimensions of which are specified by $D_1 \dots D_N$) read from the specified hardware port. Port is the hardware port number. For example,

```
result = INP(paddr)
would read a single byte, and

result = INP(paddr, 2,4)
```

would read a two-element by four-element array.

Result = INPW(Port, [$D_1 \dots D_N$])

This function returns either one 16-bit word, as an integer (if only the port number is specified), or an array (the dimensions of which are specified by $D_1 \dots D_N$) from the specified hardware port. Port is the hardware port number.

OUTP, Port, Value

This procedure writes either one byte or an array of bytes to the specified hardware port. Port is the hardware port number. *Value* is the byte value or array to be written.

OUTPW, Port, Value

This procedure writes either one 16-bit word or an array of words to the specified hardware port. Port is the hardware port number. *Value* is the integer value or array to be written.

LJLCT

This routine is obsolete and should not be used in new IDL code.

The LJLCT procedure loads standard color tables for LJ-250/252 printer. The color tables are modified only if the device is currently set to “LJ”.

The default color maps used are for the 90 dpi color palette. There are only 8 colors available at 180 dpi.

If the current device is ‘LJ’, the !D.N_COLORS system variable is used to determine how many bit planes are in use (1 to 4). The standard color map for that number of planes is loaded. These maps are described in Chapter 7 of the *LJ250/LJ252 Companion Color Printer Programmer Reference Manual*, Table 7-5. That manual gives the values scaled from 1 to 100, LJLCT scales them from 0 to 255.

This routine is written in the IDL language. Its source code can be found in the file `ljlct.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

LJLCT

Example

```
; Set plotting to the LJ device:
SET_PLOT, 'LJ'

; Load the LJ color tables:
LJLCT
```

PICKFILE

This routine is obsolete and should not be used in new IDL code.

The PICKFILE function has been renamed but retains the same functionality it had in previous releases. See `DIALOG_PICKFILE` in the *IDL Reference Guide*.

POLYFITW

This routine is obsolete and should not be used in new IDL code. To perform a weighted polynomial fit, use the **WEIGHTS** keyword to **POLY_FIT**.

The **POLYFITW** function performs a weighted least-square polynomial fit with optional error estimates and returns a vector of coefficients with a length of *NDegree*+1.

The **POLYFITW** routine uses matrix inversion. A newer version of this routine, **SVDFIT**, uses Singular Value Decomposition. The SVD technique is more flexible, but slower. Another version of this routine, **POLY_FIT**, performs a least square fit without weighting.

This routine is written in the IDL language. Its source code can be found in the file `polyfitw.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = POLYFITW(X, Y, Weights, NDegree [, Yfit, Yband, Sigma, Corrm]  
[, /DOUBLE] [, STATUS=variable] )
```

Arguments

X

An *n*-element vector of independent variables.

Y

A vector of independent variables, the same length as *X*.

Weights

A vector of weights, the same length as *X* and *Y*.

NDegree

The degree of the polynomial to fit.

Yfit

A named variable that will contain the vector of calculated *Y* values. These values have an error of plus or minus *Yband*.

Yband

A named variable that will contain the error estimate for each point.

Sigma

A named variable that will contain the standard deviation of the returned coefficients.

Corrm

A named variable that will contain the correlation matrix of the coefficients.

Keywords

DOUBLE

Set this keyword to force computations to be done in double-precision arithmetic.

STATUS

Set this keyword to a named variable to receive the status of the operation. Possible status values are:

- 0 = Successful completion.
- 1 = Singular array (which indicates that the inversion is invalid). *Result* is NaN.
- 2 = Warning that a small pivot element was used and that significant accuracy was probably lost.
- 3 = Undefined (NaN) error estimate was encountered.

Note

If STATUS is not specified, any error messages will be output to the screen.

Tip

Status values of 2 or 3 can often be resolved by setting the DOUBLE keyword.

REWIND

This routine is obsolete and should not be used in new IDL code.

The REWIND procedure rewinds the tape on the designated IDL tape unit. REWIND is available only under VMS. See the description of the magnetic tape routines in [“VMS-Specific Information”](#) in Chapter 8 of *Building IDL Applications*.

Syntax

REWIND, *Unit*

Arguments

Unit

The magnetic tape unit to rewind. *Unit* must be a number between 0 and 9, and should not be confused with standard file Logical Unit Numbers (LUNs).

RIEMANN

This routine is obsolete and should not be used in new IDL code. RIEMANN has been replaced by the [RADON](#) function.

The RIEMANN procedure computes the “Riemann sum” (or its inverse) which helps implement the backprojection operator used to reconstruct the cross-section of an object, given projections through the object from multiple directions. This technique is widely used in medical imaging in the fields of computed x-ray tomography, MRI imaging, Positron Emission Tomography (PET), and also has applications in other areas such as seismology and astronomy. The inverse Riemann sum, which evaluates the projections given a slice through an object, is also a discrete approximation to the Radon transform.

Given a matrix $A(m,n)$, which will contain the reconstructed slice; a vector P , containing the ray sums for a given view; and an angle $Theta$ measured in radians from the vertical: the Riemann sum “backprojects” the vector P into A . For each element of A , the value of the closest element of P is summed, leaving the result in A . Bilinear interpolation is an option. All operations are performed in single-precision floating point.

In the reverse operation, the ray sums contained in the view vector, P , are computed given the original slice, A , and $Theta$. This is sometimes called “front projection”.

The Riemann sum can be written:

$$\sum_{i=0}^{M-1} A(r \cdot \cos(i \cdot \Delta - \theta), i \cdot \Delta)$$

which is the sum of the data along lines through an image with an angle of θ from the vertical.

Syntax

RIEMANN, P , A , $Theta$ [, /BACKPROJECT] [, /BILINEAR] [, CENTER=*value*] [, COR=*vector*] [, CUBIC=*value*{-1 to 0}] [, D=*spacing*] [, ROW=*value*]

Arguments

P

A k -element floating-point projection vector (or matrix if the ROW keyword is specified). For backprojection (when the BACKPROJECT keyword is set), P contains the ray sums for a single view. For the inverse operation, P should contain zeros on input and will contain the ray sums for the view on output.

A

An m by n floating-point image matrix. For backprojection, A contains the accumulated results. For the inverse operation, A contains the original image. Typically, k should be larger than

$$\sqrt{m^2 + n^2}$$

which is the diagonal size of A .

Theta

The angle of the ray sums from the vertical.

Keywords

BACKPROJECT

Set this keyword to perform backprojection in which P is summed into A . If this keyword is not set, the inverse operation occurs and the ray sums are accumulated into P .

BILINEAR

Set this keyword to use bilinear interpolation rather than the default nearest neighbor sampling. Results are more accurate but slower when bilinear interpolation is used.

CENTER

Set this keyword equal to a floating-point number specifying the center of the projection. The default value for CENTER is one-half the number of elements of P .

COR

Set this keyword equal to a two-element floating-point vector specifying the center of rotation in the array A . The default value is $[m/2., n/2.]$, where A is an m by n array.

For symmetric results, given symmetric operands, COR should be set to the origin of symmetry $[(m-1)/2, (n-1)/2]$, and CENTER should be set to $(n-1)/2.$, where n is the number of elements in the projection vector, P .

CUBIC

Set this keyword to a value between -1 and 0 to use the cubic convolution interpolation method with the specified value as the interpolation parameter. Setting this keyword equal to a value greater than zero specifies a value of -1 for the interpolation parameter. Park and Schowengerdt (see reference below) suggest that a value of -0.5 significantly improves the reconstruction properties of this algorithm.

Cubic convolution is an interpolation method that closely approximates the theoretically optimum sinc interpolation function using cubic polynomials. According to sampling theory, details of which are beyond the scope of this document, if the original signal, f , is a band-limited signal, with no frequency component larger than ω_0 , and f is sampled with spacing less than or equal to $1/2\omega_0$, then f can be reconstructed by convolving with a sinc function: $\text{sinc}(x) = \sin(\pi x) / (\pi x)$.

In the one-dimensional case, four neighboring points are used, while in the two-dimensional case 16 points are used. Note that cubic convolution interpolation is significantly slower than bilinear interpolation.

For further details see:

Rifman, S.S. and McKinnon, D.M., "Evaluation of Digital Correction Techniques for ERTS Images; Final Report", Report 20634-6003-TU-00, TRW Systems, Redondo Beach, CA, July 1974.

S. Park and R. Schowengerdt, 1983 "Image Reconstruction by Parametric Cubic Convolution", Computer Vision, Graphics & Image Processing 23, 256.

D

Use this keyword to specify the spacing between elements of P , expressed in the same units as the spacing between elements of A . The default is 1.0.

ROW

Set this keyword to specify the P vector as a given row within a matrix, so that the sinogram array can be used directly without having to extract or insert each row. In this case, P must be an array with a first dimension equal to k , and the value of ROW must be in the range of 0 to the number of vectors of length k in P , minus one.

Example

This example forms a synthetic image in A , computes $Nviews$ equally spaced views, and stores the stacked projections (commonly called the “sinogram”) in a matrix PP . It then backprojects the projections into the matrix B , forming the reconstructed slice. In practical use, the projections are convolved with a filter before being backprojected.

```
; Define number of columns in A:
N = 100L

; Define number of rows in A:
M = 100L
; Number of views:
nviews = 100

; The length of the longest projection. If filtered backprojection
; is used, 1/2 the length of the convolution kernel must also be
; added.

K = CEIL(SQRT(N^2 + M^2))

; Form original slice:
A = FLTARR(N, M)

; Simulate a square object:
A[N/2:N/2+5, M/2:M/2+5] = 1.0

; Make array for sinogram:
pp = FLTARR(K, nviews)

; Compute each view:
FOR I=0, NIEWS-1 DO RIEMANN, pp, A, I * !PI/nviews, ROW=i

; Show sinogram:
TVSCL, pp

; Initial reconstructed image:
B = FLTARR(N,M)
```

```
; Do the backprojection for each view:
FOR I=0, nviews-1 DO $
    RIEMANN, pp, B, I * !PI/nviews, /BACKPROJECT, ROW=i

; Show reconstructed array:
TVSCL, B
```

RSTRPOS

This routine is obsolete and should not be used in new IDL code.

The RSTRPOS function has been replaced by the STRPOS function's REVERSE_SEARCH keyword. See “[STRPOS](#)” in the *IDL Reference Guide* manual.

The RSTRPOS function finds the *last* occurrence of a substring within an object string (the STRPOS function finds the first occurrence of a substring). If the substring is found in the expression, RSTRPOS returns the character position of the match, otherwise it returns -1.

Syntax

Result = RSTRPOS(*Expression*, *Search_String* [, *Pos*])

Arguments

Expression

The expression string in which to search for the substring.

Search_String

The substring to be searched for within *Expression*.

Pos

The character position before which the search is begun. If *Pos* is omitted, the search begins at the last character of *Expression*.

Example

```
; Define the expression:
exp = 'Holy smokes, Batman!'
; Find the position of a substring:
pos = RSTRPOS(exp, 'smokes')
; Print the substring's position:
PRINT, pos
```

IDL prints:

5

Note

Substring begins at position 5 (the sixth character).

SET_SYMBOL

This routine is obsolete and should not be used in new IDL code.

The SET_SYMBOL procedure defines a DCL (Digital Command Language) interpreter symbol for the current process. SET_SYMBOL is available only under VMS.

Syntax

SET_SYMBOL, *Name*, *Value* [, TYPE={1 | 2}]

Arguments

Name

A scalar string containing the name of the symbol to be defined.

Value

A scalar string containing the value with which *Name* is defined.

Keywords

TYPE

Indicates the table into which *Name* will be defined. Setting TYPE to 1 specifies the local symbol table, while a value of 2 specifies the global symbol table. The default is the local table.

SETLOG

This routine is obsolete and should not be used in new IDL code.

The SETLOG procedure defines a logical name.

Note

This procedure is only available for the VMS platform.

Syntax

```
SETLOG, Lognam, Value [, /CONCEALED] [, /CONFINE] [, /NO_ALIAS]  
[, TABLE=string] [, /TERMINAL]
```

Arguments

Lognam

A scalar string containing the name of the logical to be defined.

Value

A string containing the value to which the logical will be set. If *Value* is a string array, *Lognam* is defined as a multi-valued logical where each element of *Value* defines one of the equivalence strings.

Keywords

CONCEALED

If this keyword is set, RMS (VMS Record Management Services) interprets the equivalence name as a device name.

CONFINE

If this keyword is set, the logical name is not copied from the IDL process to its spawned subprocesses.

NO_ALIAS

If this keyword is set, the logical name cannot be duplicated in the same logical table at an outer access mode. If another logical name with the same name already exists at

an outer access mode, it is deleted. See the *VMS System Services Manual* for additional information on logical names and access modes.

TABLE

A scalar string containing the name of the logical table into which *Lognam* will be entered. If TABLE is not specified, LNM\$PROCESS_TABLE is used.

TERMINAL

If this keyword is set, when attempting to translate the logical, further iterative logical name translation on the equivalence name is not to be performed.

SIZE Executive Command

This command is obsolete and is should not be used in new IDL code.

.SIZE Code_Size, Data_Size

The `.SIZE` command resizes the memory area used to compile programs. The default code and data area sizes are 32,768 and 8,192 bytes, respectively. These sizes represent a compromise between an unlimited program space and conservation of memory. User procedures and functions are compiled in this large program area. After successful compilation, a new memory area of the required size is allocated to contain the newly compiled program unit.

Resizing the code and data areas erases the currently compiled main program and all main program variables. For example, to extend the code and data areas to 30,000 and 5,000 bytes, respectively, use the following statement:

```
.SIZE 30000 5000
```

Each user-defined procedure, function, and main program has its own code area that contains the compiled code and constants. Although the maximum size of these areas is set by the `.SIZE` command, there is virtually no limit to the number of program units. Procedures or functions that run out of code area space should be broken into multiple program units.

The data area contains information describing the user-defined variables and common blocks for each procedure, function, or main program. Note that the “data area” is not the space available for variable storage, but the space available for that program unit’s symbol table.

Warning

Users are sometimes confused about the nature of the code and data areas. Note that there are separate code and data areas for each compiled function, routine, or main program. The `HELP` command can be used to see the current sizes of the code and data areas for the program unit in which the `HELP` function is called.

For example, to see the sizes of the code and data areas for the main program level, enter the following at the IDL prompt:

```
HELP
```

Each compiled function and procedure has its own code and data areas. If the compiled routine does not use the full amount of code space allocated by the default

code area size, the code area “shrinks” to just the size the routine needs. For example, enter and compile a simple procedure from the IDL prompt by entering:

```
.RUN
- PRO EXAMPLE
- PRINT, "Here are the code and data areas for this procedure:"
- HELP
- END
```

Call the EXAMPLE procedure from the command line to see the result:

```
EXAMPLE
```

The third line of output from the HELP procedure displays:

```
Code area used: 100.00% (100/100), Data area used: 2.02% (2/99)
```

Note that the code area for the EXAMPLE procedure is completely filled and that the total size of the code area is just 100 bytes.

SKIPF

This routine is obsolete and should not be used in new IDL code.

The SKIPF procedure skips records or files on the designated magnetic tape unit. SKIPF is available only under VMS. If two parameters are supplied, files are skipped. If three parameters are present, individual records are skipped.

The number of files or records actually skipped is stored in the system variable !ERR. Note that when skipping records, the operation terminates immediately when the end of a file is encountered. See the description of the magnetic tape routines in [“VMS-Specific Information”](#) in Chapter 8 of *Building IDL Applications*.

Syntax

SKIPF, *Unit*, *Files*

or

SKIPF, *Unit*, *Records*, *R*

Arguments

Unit

The magnetic tape unit to rewind. *Unit* must be a number between 0 and 9, and should not be confused with the standard file Logical Unit Numbers (LUNs).

Files

The number of files to be skipped. Skipping is in the forward direction if the second parameter is positive, otherwise files are skipped backwards.

Records

The number of records to be skipped. Skipping is in the forward direction if the second parameter is positive, otherwise records are skipped backwards.

R

If this argument is present, records are skipped, otherwise files are skipped. The value of *R* is never examined. Its presence serves only to indicate that records are to be skipped.

SLICER

This routine is obsolete and should not be used in new IDL code.

The IDL SLICER is a widget-based application to show 3D volume slices and isosurfaces. On exit, the Z-buffer contains the most recent image generated by the SLICER. The image may be redisplayed on a different device by reading the Z-buffer contents plus the current color table. Note that the volume data must fit in memory.

Using the SLICER

Data is passed to the SLICER via the common block VOLUME_DATA. Note that the variable used to contain the volume data must be defined as part of the common block *before* the volume data is read into the variable. (See the *Example* section, below.)

The SLICER has the following modes:

- Slices: Displays or removes orthogonal or oblique slices through the data volume.
- Block: Displays the surfaces of a selected block inside the volume.
- Cutout: Cuts blocks from previously drawn objects.
- Isosurface: Draws an isosurface contour.
- Probe: Displays the position and value of objects using the mouse.
- Colors: Manipulates the color tables and contrast.
- Rotations: Sets the orientation of the display.
- Journal: Records or plays back files of SLICER commands.

See the SLICER's help file (available by clicking the "Help" button on the SLICER widget) for more information about drawing slices and images.

Syntax

```
COMMON VOLUME_DATA, A
```

```
A = your_volume_data
```

```
SLICER
```

Arguments

A

A 3D array containing volume data. Note that the variable *A* must be included in the common block `VOLUME_DATA` *before* being equated with the volume data. *A* is *not* an explicit argument to `SLICER`.

Keywords

CMD_FILE

Set this keyword to a string that contains the name of a file containing `SLICER` commands to execute as described under *SLICER Commands*, below. The file should contain one command per line.

Command files can be created interactively, using the `SLICER`'s "Journal" feature.

COMMAND

Set this keyword equal to a $1 \times n$ string array containing commands to be executed by the `SLICER` before entering interactive mode. Available commands are described under *SLICER Commands*, below.

Note that commands passed to the `SLICER` with the `COMMAND` keyword must be in a $1 \times n$ array, rather than in an n -element vector. String arrays can be easily specified in the proper format using the `TRANSPOSE` command. For example, the following passes three commands to the slicer:

```
com=TRANSPOSE(['COLOR 5', 'TRANS 1 20', 'ISO 17 1'])
SLICER, COMMAND=com
```

DETACHED

Set this keyword to put the drawable in a separate window. This can be useful when working with large images.

GROUP

Set this keyword to the widget ID of the widget that calls `SLICER`. When `GROUP` is specified, a command to destroy the calling widget also destroys the `SLICER`.

NO_BLOCK

Set this keyword equal to zero to have XMANAGER *block* when this application is registered. By default, NO_BLOCK is set equal to one, providing access to the command line if active command line processing is available. Setting NO_BLOCK=0 will cause *all* widget applications to block, not just this application. For more information, see the documentation for the NO_BLOCK keyword to XMANAGER.

RANGE

Set this keyword to a two-element array containing minimum and maximum data values of interest. If RANGE is omitted, the data is scanned for the minimum and maximum values.

RESOLUTION

Set this keyword to a two-element vector specifying the width and height of the drawing window. The default is 55% by 44% of the screen width.

SLICER Commands

The slicer accepts a number of commands that replicate the action of controls in the graphical user interface. These commands can be specified at the IDL command line using either CMD_FILE keyword or the COMMAND keyword. Files of SLICER commands can also be created and played back from within the SLICER, using the “Journal” feature.

Commands, in this context, are strings that include a command identifier and (in some cases) one or more numeric parameters separated by blanks. The following are the available SLICER commands, with parameters.

COLOR *Table_Index Low High Shading*

Set the color tables. *Table_Index* is the pre-defined color table number (see LOADCT), or -1 to retain the present table. *Low* is the contrast minimum, *High* is the contrast maximum, and *Shading* is the differential shading, all expressed in percent. For example, the following command picks color table number 2, sets the minimum contrast to 10%, the maximum contrast to 90%, and the differential shading to 50%:

```
COLOR 2 10 90 50
```

CUBE *Mode Cut_Ovr Interp X0 Y0 Z0 X1 Y1 Z1*

Defines the volume used for “Block” and “Cutout” operations. Set *Mode*=1 for Block mode or *Mode*=2 for Cutout mode. Set *Cut_Ovr*=0 to mimic selecting the “Cut Into” button or *Cut_Ovr*=1 to mimic selecting the “Cut Over” button.

Note

These buttons have no effect in Block mode. See the online help on SLICER for further explanation of Cut Into and Cut Over.

Set *Interp*=1 for bilinear interpolation sampling or *Interp*=0 for nearest neighbor sampling.

X0,Y0,Z0 are the coordinates of the lower corner of the volume, and *X1,Y1,Z1* are the coordinates of the upper corner. For example:

```
CUBE 1 0 1 20 0 56 60 75 42
```

selects Block mode, the “Cut Into” button, bilinear interpolation and defines the volume’s corners at (20, 0, 56) and (60, 75, 42).

ERASE

Erases the display. Mimics clicking on the “Erase” button.

ISO *Threshold Hi_Lo*

Draws an iso-surface. *Threshold* is the isosurface threshold value. Set *Hi_Lo* equal to 1 to view the low side, or equal to 0 to view the high side.

ORI *X_Axis Y_Axis Z_axis X_Rev Y_Rev Z_Rev X_Rot Z_Rot Asp*

Sets the orientation for the SLICER display, mimicking the action of the “Orientation” button. Set *X_Axis*, *Y_Axis*, and *Z_Axis* to 0, 1, or 2, where 0 represents the data X axis, 1 the data Y axis, and 2 the data Z axis. Set *X_Rev*, *Y_Rev*, and *Z_Rev* to 0 for normal orientation or to 1 for reversed. Set *X_Rot* and *Z_Rot* to the desired rotations of the X and Z axes, in degrees (30 is the default). Set *Asp* to the desired Z axis aspect ratio with respect to X and Y. For example, to interchange the X and Z axes and reverse the Y use the string:

```
ORI 2 1 0 0 1 0 30 30 1
```

SLICE *Axis Value Interp Expose 0*

Draws an orthogonal slice. Set *Axis* to 0 to draw a slice parallel to the X axis, to 1 for the Y axis, or to 2 for the Z axis. Set *Value* to the pixel value of the slice. Set *Interp*=1

for bilinear interpolation sampling or *Interp*=0 for nearest neighbor sampling. Set *Expose*=1 to cut out of an existing image (mimicking the “Expose” button) or set *Expose*=0 to draw the slice on top of the current display (mimicking the “Draw” button). The final zero indicates that the slice is orthogonal rather than oblique. For example, the following command draws an orthogonal slice parallel to the X axis, at the pixel value 31, using bilinear interpolation.

```
SLICE 0 31 1 0 0
```

SLICE *Azimuth Elev Interp Expose 1 X0 Y0 Z0*

Draws an oblique slice. The oblique plane crosses the XY plane at angle *Azimuth*, with an elevation of *Elev*. Set *Interp*=1 for bilinear interpolation sampling or *Interp*=0 for nearest neighbor sampling. Set *Expose*=1 to cut out of an existing image (mimicking the “Expose” button) or set *Expose*=0 to draw the slice on top of the current display (mimicking the “Draw” button). The one indicates that the slice is oblique rather than orthogonal. The plane passes through the point (*X0*, *Y0*, *Z0*). For example, the following command exposes an oblique slice with an azimuth of 42 and an elevation of 24, using bilinear interpolation. The plane passes through the point (52, 57, 39).

```
SLICE 42 24 1 1 1 52 57 39
```

TRANS *On_Off Threshold*

Turns transparency on or off and sets the transparency threshold value. Set *On_Off*=1 to turn transparency on, *On_Off*=0 to turn transparency off. *Threshold* is expressed in percent of data range (0 = minimum data value, 100 = maximum data value). For example, this command turns transparency on and sets the threshold at 20 percent:

```
TRANS 1 20
```

UNDO

Undoes the previous operation.

WAIT Secs

Causes the SLICER to pause for the specified time, in seconds.

Example

Data is transferred to the SLICER via the `VOLUME_DATA` common block instead of as an argument. This technique is used because volume datasets can be very large

and the duplication that occurs when passing values as arguments is a waste of memory.

Suppose that you want to read some data from the file `head.dat`, which is included in the IDL examples directory, into IDL for use in the SLICER. Before you read the data, establish the `VOLUME_DATA` common block with the following command:

```
COMMON VOLUME_DATA, VOL
```

The `VOLUME_DATA` common block has just one variable in it. (The variable can have any name; here, we're using the name `VOL`.) Now read the data from the file into `VOL`. For example:

```
OPENR, UNIT, /GET, FILEPATH('head.dat', SUBDIRECTORY=[ 'examples',
'data'])
VOL = BYTARR(80, 100, 57, /NOZERO)
READU, UNIT, VOL
CLOSE, UNIT
```

Now you can run the SLICER widget application by entering:

```
SLICER
```

The data stored in `VOL` is the data being worked on by the SLICER.

To obtain the image in the slicer window after slicer is finished:

```
SET_PLOT, 'Z' Use the Z buffer graphics device.
A = TVRD() Read the image.
```

STR_SEP

This routine is obsolete and should not be used in new IDL code.

The STR_SEP function has been replaced by STRSPLIT for single character delimiters, and STRSPLIT with the REGEX keyword set for longer delimiters. See “STRSPLIT” in the *IDL Reference Guide* manual.

The STR_SEP function divides a string into pieces as designated by a separator string. STR_SEP returns a string array where each element is a separated piece of the original string.

Syntax

Result = STR_SEP(*Str*, *Separator* [, /TRIM] [, /REMOVE_ALL] [, /ESC])

Arguments

Str

The string to be separated.

Separator

The separator string.

Keywords

TRIM

Set this keyword to remove leading and trailing blanks from each element of the returned string array. TRIM performs STRTRIM(*String*, 2).

REMOVE_ALL

Set this keyword to remove all blanks from each element of the returned string array. REMOVE_ALL performs STRCOMPRESS(*String*, /REMOVE_ALL)

ESC

Set this keyword to interpret the characters following the <ESC> character literally and not as separators. For example, if the separator is a comma and the escape

character is a backslash, the character sequence “a\b” is interpreted as a single field containing the characters “a,b”.

Example

```
; Create a string:
str = 'Doug.is.a.cool.dude!'

; Separate the parts between the periods:
parts = STR_SEP(str, '.')

; Confirm that the string has been broken up into 5 elements:
HELP, parts

PRINT, parts[3]
```

IDL Output

```
PARTS    STRING = Array[5]
cool
```

TAPRD

This routine is obsolete and should not be used in new IDL code.

The TAPRD procedure reads the next record on the selected tape unit into the specified array. TAPRD is available only under VMS. No data or format conversion, with the exception of optional byte reversal, is performed. The array must be defined with the desired type and dimensions. If the read is successful, the system variable !ERR is set to the number of bytes read. See the description of the magnetic tape routines in “[VMS-Specific Information](#)” in Chapter 8 of *Building IDL Applications*.

Syntax

TAPRD, *Array*, *Unit* [, *Byte_Reverse*]

Arguments

Unit

The magnetic tape unit to read. This argument must be a number between 0 and 9, and should not be confused with standard file Logical Unit Numbers (LUN's).

Array

A named variable into which the data is read. If *Array* is larger than the tape record, the extra elements of the array are not changed. If the array is shorter than the tape record, a data overrun error occurs. The length of *Array* and the records on the tape can range from 14 bytes to 65,235 bytes.

Byte_Reverse

If this parameter is present, the even and odd numbered bytes are swapped after reading, regardless of the type of data or variables. This enables reading tapes containing short integers that were written on machines with different byte ordering. You can also use the BYTORDER routine to re-order different data types.

TAPWRT

This routine is obsolete and should not be used in new IDL code.

The TAPWRT procedure writes data from the *Array* parameter to the selected tape unit. TAPWRT is available only under VMS. One physical record containing the same number of bytes as the array is written each time TAPWRT is called. The parameters and usage are identical to those in the TAPRD procedure with the exception that here the *Array* parameter can be an expression. Consult the TAPRD procedure for details. See the description of the magnetic tape routines in “[VMS-Specific Information](#)” in Chapter 8 of *Building IDL Applications*.

Syntax

```
TAPWRT, Array, Unit [, Byte_Reverse]
```

Arguments

Unit

The magnetic tape unit to write. This argument must be a number between 0 and 9, and should not be confused with standard file Logical Unit Numbers (LUNs).

Array

The expression representing the data to be output. The length of *Array* and the records on the tape can range from 14 bytes to 65,235 bytes.

Byte_Reverse

If this parameter is present, the even and odd numbered bytes are swapped on output, regardless of the type of data or variables. This enables writing tapes that are compatible with other machines.

TIFF_DUMP

This routine is obsolete and should not be used in new IDL code.

The TIFF_DUMP procedure dumps the Image File Directories of a TIFF file directly to the terminal screen. Each TIFF Image File Directory entry is printed. This procedure is used mainly for debugging.

Note that not all of the tags have names encoded. In particular, Facsimile, Document Storage and Retrieval, and most no-longer-recommended fields are not encoded.

Syntax

TIFF_DUMP, *File*

Arguments

File

A scalar string containing the name of file to read.

TIFF_READ

This routine is obsolete and should not be used in new IDL code.

The TIFF_READ function has been renamed but retains the same functionality it had in previous releases. See READ_TIFF in the *IDL Reference Guide*.

The TIFF_READ function reads 8-bit or 24-bit images in TIFF format files (classes G, P, and R) and returns the image and color table vectors in the form of IDL variables. Only one image per file is read. TIFF_READ returns a byte array containing the image data. The dimensions of the result are the same as defined in the TIFF file (*Columns*, *Rows*).

For TIFF images that are RGB interleaved by pixel, the output dimensions are (3, *Columns*, *Rows*).

For TIFF images that are RGB interleaved by image, TIFF_READ returns the integer value zero, sets the variable defined by the PLANARCONFIG keyword to 2, and returns three separate images in the variables defined by the R, G, and B arguments.

Syntax

Result = TIFF_READ(*File* [, *R*, *G*, *B*])

Arguments

File

A scalar string containing the name of file to read.

R, G, B

Named variables that will contain the Red, Green, and Blue color vectors extracted from TIFF Class P, Palette Color images. For TIFF images that are RGB interleaved by image (when the variable specified by the PLANARCONFIG keyword is returned as 2) the R, G, and B variables each hold an image with the dimensions (*Columns*, *Rows*).

Keywords

ORDER

Set this keyword to a named variable that will contain the order parameter from the TIFF File. This parameter is returned as 0 for images written bottom to top, and 1 for images written top to bottom. If the Orientation parameter does not appear in the TIFF file, an order of 1 is returned.

PLANARCONFIG

Set this keyword to a named variable that will contain the interleave parameter from the TIFF file. This parameter is returned as 1 for TIFF files that are GrayScale, Palette, or RGB color interleaved by pixel, or as 2 for RGB color TIFF files interleaved by image.

Example

Read the file `my.tif` in the current directory into the variable `image`, and save the color tables in the variables, `R`, `G`, and `B` by entering:

```
image = TIFF_READ('my.tif', R, G, B)
```

To view the image, load the new color table and display the image by entering:

```
TVLCT, R, G, B  
TV, image
```

TIFF_WRITE

This routine is obsolete and should not be used in new IDL code.

The TIFF_WRITE procedure has been renamed but retains the same functionality it had in previous releases. See WRITE_TIFF in the *IDL Reference Guide*.

The TIFF_WRITE procedure writes 8- or 24-bit images to a TIFF file. Files are written in a single strip, or in three strips when the PLANARCONFIG keyword is set to 2.

Syntax

TIFF_WRITE, *File*, *Array* [, *Orientation*]

Arguments

File

A scalar string containing the name of file to create.

Array

The image data to be written. If not already a byte array, it is made a byte array. Array may be either an (n, m) array for Grayscale or Palette classes, or a $(3, n, m)$ array for RGB full color, interleaved by image. If the PLANARCONFIG keyword is set to 2 then the *Array* parameter is ignored (and may be omitted).

Orientation

This parameter should be 0 if the image is stored from bottom-to-top (the default). For images stored from top-to-bottom, this parameter should be 1.

Warning: not all TIFF readers are capable of reversing the scan line order. If in doubt, first convert the image to top-to-bottom order (use the REVERSE function), and set *Orientation* to 1.

Keywords

RED, GREEN, BLUE

If you are writing a Class P, Palette color image, set these keywords equal to the color table vectors, scaled from 0 to 255.

If you are writing an image that is RGB interleaved by image (i.e., if the `PLANARCONFIG` keyword is set to 2), set these keywords to the names of the variables containing the 3 color component image.

PLANARCONFIG

Set this keyword to 2 if writing an RGB image that is contained in three separate images (color planes). The three images must be stored in variables specified by the `RED`, `GREEN`, and `BLUE` keywords. Otherwise, omit this parameter (or set it to 1).

XRESOL

The horizontal resolution, in pixels per inch. The default is 100.

YRESOL

The vertical resolution, in pixels per inch. The default is 100.

Examples

Four types of TIFF files can be written:

TIFF Class G, Grayscale.

The variable `array` contains the 8-bit image array. A value of 0 is black, 255 is white. The `Red`, `Green`, and `Blue` keywords are omitted.

```
TIFF_WRITE, 'a.tif', array
```

TIFF Class P, Palette Color

The variable `array` contains the 8-bit image array. The keyword parameters `RED`, `GREEN`, and `BLUE` contain the color tables, which can have up to 256 elements, scaled from 0 to 255.

```
TIFF_WRITE, 'a.tif', array, RED = r, GREEN = g, BLUE = b
```

TIFF Class R, RGB Full Color, color interleaved by pixel

The variable `array` contains the byte data, and is dimensioned (`3, cols, rows`).

```
TIFF_WRITE, 'a.tif', array
```

TIFF Class R, RGB Full Color, color interleaved by image

Input is three separate images, provided in the keyword parameters `RED`, `GREEN`, and `BLUE`. The input argument `Array` is ignored. The keyword `PLANARCONFIG` must be set to 2 in this case.

```
TIFF_WRITE, 'a.tif', RED = r, GREEN = g, BLUE = b, PLAN = 2
```

TRNLOG

This routine is obsolete and should not be used in new IDL code.

The TRNLOG function searches the VMS logical name tables for a specified logical name and returns the equivalence string(s) in an IDL variable. TRNLOG is available only under VMS. TRNLOG also returns the VMS status code associated with the translation as a longword value. As with all VMS status codes, success is indicated by an odd value (least significant bit is set) and failure by an even value.

Syntax

```
Result = TRNLOG( Lognam, Value [, ACMODE={0 | 1 | 2 | 3}]  
[, /FULL_TRANSLATION] [, /ISSUE_ERROR] [, RESULT_ACMODE=variable]  
[, RESULT_TABLE=variable] [, TABLE=string] )
```

Arguments

Lognam

A scalar string containing the name of the logical to be translated.

Value

A named variable into which the equivalence string is placed. If Lognam has more than one equivalence string, the first one is used. The FULL_TRANSLATION keyword can be used to obtain all equivalence strings.

Keywords

ACMODE

Set this keyword to a value specifying the access mode to be used in the translation. Valid values are:

- 0 = Kernal
- 1 = Executive
- 2 = Supervisor
- 3 = User

When you specify the ACMODE keyword, all names at access modes less privileged than the specified mode are ignored. If you do not specify ACMODE, the translation

proceeds without regard to access mode. However, the search proceeds from the outermost (User) to the innermost (Kernal) mode. Thus, if two logical names with the same name but different access modes exist in the same table, the name with the outermost access mode is used.

FULL_TRANSLATION

Set this keyword to obtain the full set of equivalence strings for *Lognam*. By default, when translating a multivalued logical name, *Value* only receives the first equivalence string as a scalar value. When this keyword is set, *Value* instead returns a string array. Each element of this array contains one of the equivalence strings. For example, under recent versions of VMS, the SYSSYSROOT logical can have multiple values. To see these values from within IDL, enter:

```
; Translate the logical:
ret = TRNLOG('SYSSYSROOT', trans, /FULL, /ISSUE_ERROR)
; View the equivalence strings:
PRINT, trans
```

ISSUE_ERROR

Set this keyword to issue an error message if the translation fails. Normally, no error is issued and the user must examine the return value to determine if the operation failed.

RESULT_ACMODE

If present, this keyword specifies a named variable in which to place the access mode of the translated logical. The access modes are summarized above.

RESULT_TABLE

If present, this keyword specifies a named variable. The name of the logical table containing the translated logical is placed in this variable as a scalar string.

TABLE

A scalar string giving the name of the logical table in which to search for *Lognam*. If TABLE is not specified, the standard VMS logical tables are searched until a match is found, starting with LNM\$PROCESS_TABLE and ending with LNM\$SYSTEM_TABLE.

VAX_FLOAT

This routine is obsolete and should not be used in new IDL code.

The VAX_FLOAT function performs one of two possible actions:

1. Determine, and optionally change, the default value for the VAX_FLOAT keyword to the OPEN procedures.
2. Determine if an open file unit has the VAX_FLOAT attribute set.

Syntax

Result = VAX_FLOAT([*Default*] [, FILE_UNIT=*lun*])

Arguments

Default

Default is used to change the default value of the VAX_FLOAT keyword to the OPEN procedures. A value of 0 (zero) makes the default for those keywords False. A non-zero value makes the default True. Specifying *Default* in conjunction with the FILE_UNIT keyword will cause an error.

Note

If the FILE_UNIT keyword is *not* specified, the value returned from VAX_FLOAT is the default value *before* any change is made. This is the case even if *Default* is specified. This allows you to get the old setting and change it in a single operation.

Keywords

FILE_UNIT

Set this keyword equal to the logical file unit number (LUN) of an open file. VAX_FLOAT returns True (1) if the file was opened with the VAX_FLOAT attribute, or False (0) otherwise. Setting the FILE_UNIT keyword when the *Default* argument is specified will cause an error.

Example

To determine if the default VAX_FLOAT keyword value for OPEN is True or False:

```
default_vax_float = VAX_FLOAT()
```


To determine the current default value of the `VAX_FLOAT` keyword for `OPEN` and change it to `True` (1) in a single operation:

```
old_vax_float = VAX_FLOAT(1)
```

To determine if the file currently open on logical file unit 1 was opened with the `VAX_FLOAT` keyword set:

```
file_is_vax_float = VAX_FLOAT(FILE_UNIT=1)
```

WEOF

This routine is obsolete and should not be used in new IDL code.

The WEOF procedure writes an end of file mark, sometimes called a tape mark, on the designated tape unit at the current position. WEOF is available only under VMS. The tape must be mounted as a foreign volume.

Syntax

WEOF, *Unit*

Arguments

Unit

The magnetic tape unit on which the end of file mark is written. This argument must be a number between 0 and 9, and should not be confused with standard file Logical Unit Numbers (LUNs).

WIDED

This routine is obsolete and should not be used in new IDL code.

The WIDED procedure invokes IDL's graphical user interface designer, known as the Widget Builder. This functionality has been replaced by the GUIBuilder, which is documented in *Building IDL Applications*.

Syntax

WIDED

WIDGET_MESSAGE

This routine is obsolete and should not be used in new IDL code.

The WIDGET_MESSAGE function has been renamed but retains the same functionality it had in previous releases. See “[DIALOG_MESSAGE](#)” in the *IDL Reference Guide* manual.



Chapter 3:

Routines with Obsolete Keywords

This chapter contains documentation for keywords that have been removed from IDL routines. New IDL code should not use these keywords. See [Appendix O, “Obsolete Features”](#) in the *IDL Reference Guide* manual for a list of obsolete keywords and their replacements, if suitable replacements exist.

When IDL attempts to execute a routine called with an obsolete keyword, one of the following things will happen:

1. The routine may function as originally designed, with no change in behavior. This is often the case when the obsolete keyword has been replaced by another keyword with a more efficient or slightly different mechanism. In these cases, the obsolete keyword is generally re-implemented within the routine to use the mechanism of the new keyword, allowing code that uses the obsolete keyword to run unaltered. Note that although the results will be the same as before the keyword became obsolete, the code may run more efficiently if the replacement keyword is used instead of the obsolete keyword.

Example: The GROUP keyword to the DIALOG_PICKFILE routine was replaced by the DIALOG_PARENT keyword. Code that uses the GROUP keyword continues to run as it always did.

2. The routine may quietly accept the keyword, but ignore its presence. This is the case when the presence of the obsolete keyword does not change the result returned by the routine. For example, keywords that affected attributes only available on certain platforms may simply be ignored on other platforms. Code using obsolete keywords of this type can run unaltered.

Example: The MACTYPE keyword to the OPEN routine changed an attribute of files on pre-OS X Macintosh filesystems that has no corollary on other filesystems. IDL simply ignores the presence of this keyword.

3. The routine may generate an error. This is the case when the presence of the obsolete keyword changes the result returned by the routine. For example, keywords that affected the returned data in some way that is no longer supported must now be removed from IDL code before it will run.

Example: The DTOGFLOAT keyword to the BYTEORDER routine converted data to a format only supported under VMS. The underlying mechanism used is not available in other operating systems, and IDL will generate an error if such a conversion is specified in the call to BYTEORDER.

In all cases, if IDL code containing calls to obsolete keywords compiles and runs without error, the results are the same as they would have been before the keyword was made obsolete.

BYTEORDER

The following keywords to the BYTEORDER procedure are obsolete.

VMS-Only Keywords

DTOGFLOAT

Set this keyword to convert native (IEEE) double-precision floating-point format to VAX G float format. Note that IDL does not support the VAX G float format via any other mechanism.

GFLOATTOD

Set this keyword to convert VAX G float format to native (IEEE) double-precision floating-point format. Note that IDL does not support the VAX G float format via any other mechanism.

CALL_EXTERNAL

The following keywords to the CALL_EXTERNAL function are obsolete.

Keywords

DEFAULT

This keyword is ignored on non-VMS platforms. Under VMS, it is a string containing the default device, directory, file name, and file type information for the file that contains the sharable image.

PORTABLE

Under VMS, causes CALL_EXTERNAL to use the IDL Portable calling convention for passing arguments to the called function instead of the default VMS LIB\$CALLG convention. Under other operating systems, only the portable convention is available, so this keyword is quietly ignored.

If you are using the IDL Portable calling convention, the AUTO_GLUE or WRITE_WRAPPER keywords are available to simplify the task of matching the form in which IDL passes the arguments to the interface of your target function.

VAX_FLOAT (VMS Only)

If specified, all data passed to the called function is first converted to VAX F (single) or D (double) floating point formats. On return, any data passed by reference is converted back to the IEEE format used by IDL. This feature allows you to call code compiled to work with earlier versions of IDL, which used the old VAX formats.

The default setting for this keyword is FALSE, unless IDL was started with the VAX_FLOAT startup option, in which case the default is TRUE. See “Command Line Options” in Chapter 4 of *Using IDL* for details on this qualifier. You can change this setting at runtime using the VAX_FLOAT function.

DEVICE

The following keywords to the DEVICE procedure are obsolete.

Keywords

DEPTH

(LJ)

The DEPTH keyword specifies the number of significant bits in a pixel. The LJ250 can support between 1 and 4 significant bits (known also as planes). The number of available colors is related to the number of significant planes by the equation:

$$\text{Colors} = 2^{\text{\#planes}}$$

Therefore, the LJ250 can support 2, 4, 8, or 16 separate colors on a single page of output. The default is to use a single plane, producing monochrome output.

Since IDL is based around 8-bit pixels, it is necessary to define which bits in a 8-bit pixel are used by the LJ250 driver, and which are ignored. When using a depth of 1 (monochrome), dithering techniques are used to render images. In this case, all 8 bits are used. If more than a single plane is used, the least significant *n* bits of a 8-bit pixel are used, where *n* is the selected depth. For example, using a depth of 4, pixel values of 15, 31, and 47 are all considered to have the value 15 because all three values have the same binary representation in their 4 least significant digits.

When the depth is changed, the standard color map given in Table 7-5 of the *LJ250/LJ252 Companion Color Printer Programmer Reference Manual* is automatically loaded. Therefore, color maps should be loaded with TVLCT after changing the depth.

FONT

(WIN, X)

This keyword is now obsolete and has been replaced by the [SET_FONT](#) keyword. Code that uses the FONT keyword will continue to function as before, but we suggest that all new code use SET_FONT.

DOC_LIBRARY

The following keywords to the DOC_LIBRARY procedure are obsolete.

VMS Keywords

FILE

If this keyword is set, the output is left in the file `userlib.doc`, in the current directory.

PATH

A string that describes an optional directory/library search path. This keyword uses the same format and semantics as `!PATH`. If omitted, `!PATH` is used.

OUTPUTS

If this keyword is set, documentation is sent to the standard output unless the `PRINT` keyword is set.

EXTRACT_SLICE

The following keywords to the EXTRACT_SLICE procedure are obsolete.

CUBIC

Set this keyword to use cubic interpolation. The default is to use tri-linear interpolation. If the SAMPLE keyword is set, then the CUBIC keyword is ignored.

IDLgrMPEG::Save

The following keywords to the IDLgrMPEG::Save procedure method are obsolete.

Keywords

CREATOR_TYPE

Set this keyword to a four character string representing the creator string to be used when writing this file on a Macintosh. This property is ignored if the current platform is not a Macintosh. The default is TVOD (Apple Movie Player application).

IDLgrVolume::Init

The following keywords to the IDLgrVolume::Init procedure method are obsolete.

Keywords

CUTTING_PLANES (*Get*, *Set*)

Set this keyword to a floating-point array with dimensions (4, n) specifying the coefficients of n cutting planes. The cutting plane coefficients are in the form $\{ \{n_x, n_y, n_z, D\}, \dots \}$ where $(n_x)X + (n_y)Y + (n_z)Z + D > 0$, and (X, Y, Z) are the voxel coordinates. To clear the cutting planes, set this property to any scalar value (e.g. CUTTING_PLANES = 0). By default, no cutting planes are defined.

LINKIMAGE

The following keywords to the LINKIMAGE procedure are obsolete.

Keywords

DEFAULT

This keyword is ignored on non-VMS platforms. Under VMS, it is a string containing the default device, directory, file name, and file type information for the file that contains the sharable image. See [“VMS LINKIMAGE and LIB\\$FIND_IMAGE_SYMBOL”](#) on page 1281 for additional information.

LIVE_PRINT

The following keywords to the LIVE_PRINT procedure are obsolete.

Keywords

SETUP

(Macintosh users only) Set this keyword to have a printer setup dialog appear. This keyword allows the user to setup the page for printing.

MAKE_DLL

The following keywords to the MAKE_DLL procedure are obsolete.

VMS-Only Keywords

This keyword is for VMS platforms only, and is ignored on all other platforms.

VAX_FLOAT

If set, specifies the sharable library to be compiled for VAX F (single) or D (double) floating point formats. The default is to use the IEEE format used by IDL.

ONLINE_HELP

The following keywords to the three `ONLINE_HELP` procedure are obsolete.

HTML_HELP

Set this keyword to a non-zero value to indicate that the file specified by the `BOOK` keyword should be viewed with the HTML Help viewer. Explicitly set this keyword equal to zero to indicate that the file should be viewed with the traditional Windows help viewer.

Note

Normally, `ONLINE_HELP` can properly determine which viewer to use based on the name of the file, so use of the `HTML_HELP` keyword is rarely necessary.

OPEN

The following keywords to the three OPEN procedures are obsolete.

Macintosh-Only Keywords

MACCREATOR

Use this keyword to specify a four-character scalar string identifying the Macintosh file creator code of the file being created. For example, set

```
MACCREATOR = 'MSWD'
```

to create a file with the creator code MSWD. The default creator code is MIDL.

MACTYPE

Use this keyword to specify a four-character scalar string identifying the Macintosh file type of the file being created. For example, set

```
MACTYPE = 'PICT'
```

to create a file of type PICT. The default file type is TEXT.

UNIX-Only Keywords

The previous keyword NOSTDIO is now obsolete. It has been renamed RAWIO to reflect the fact that stdio may or may not actually be used. All references to NOSTDIO should be changed to be RAWIO, but NOSTDIO will still be accepted as a synonym for RAWIO.

NOSTDIO

Set this keyword to disable all use of the standard UNIX I/O for the file, in favor of direct calls to the operating system. This allows direct access to devices, such as tape drives, that are difficult or impossible to use effectively through the standard I/O.

Using this keyword has the following implications:

- No formatted or associated (ASSOC) I/O is allowed on the file. Only READU and WRITEU are allowed.
- Normally, attempting to read more data than is available from a file causes the unfilled space to be set to zero and an error to be issued. This does not happen with files opened with NOSTDIO. When using NOSTDIO, the programmer

must check the transfer count, either via the `TRANSFER_COUNT` keywords to `READU` and `WRITEU`, or the `FSTAT` function.

- The `EOF` and `POINT_LUN` functions cannot be used with a file opened with `NOSTDIO`.
- Each call to `READU` or `WRITEU` maps directly to UNIX `read(2)` and `write(2)` system calls. The programmer must read the UNIX system documentation for these calls and documentation on the target device to determine if there are any special rules for I/O to that device. For example, the size of data that can be transferred to many cartridge tape drives is often forced to be a multiple of 512 bytes.

VMS-Only Keywords

BLOCK

Set this keyword to process the file using RMS block mode. In this mode, most RMS processing is bypassed and IDL reads and writes to the file in disk block units. Such files can only be accessed via unformatted I/O commands. Block mode files are treated as an uninterpreted stream of bytes in a manner similar to UNIX stream files.

For best performance, by default IDL uses RMS block mode for fixed length record files. However, when the `SHARED` keyword is present, IDL uses standard RMS mode. Do not specify both `BLOCK` and `SHARED`.

This keyword is ignored when used with stream files.

Note

With some controller/disk combinations, RMS does not allow transfer of an odd number of bytes.

DEFAULT

A scalar string that provides a default file specification from which missing parts of the `File` argument are taken. For example, to make `.LOG` be the default file extension when opening a new file, use the command:

```
OPENW, 'DATA', DEFAULT='.LOG'
```

This statement will open the file `DATA.LOG`.

EXTENDSIZE

File extension is a relatively slow operation, and it is desirable to minimize the number of times it is done. In order to avoid the unacceptable performance that would result from extending a file a single block at a time, VMS extends its size by a default number of blocks in an attempt to trade a small amount of wasted disk space for better performance. The `EXTENDSIZE` keyword overrides the default, and specifies the number of disk blocks by which the file should be extended. This keyword is often used in conjunction with the `INITIALSIZE` and `TRUNCATE_ON_CLOSE` keywords.

FIXED

Set this keyword to indicate that the file has fixed-length records. The *Record_Length* argument is required when opening new, fixed-length files.

FORTTRAN

Set this keyword to use FORTRAN-style carriage control when creating a new file. The first byte of each record controls the formatting.

INITIALSIZE

The initial size of the file allocation in blocks. This keyword is often used in conjunction with the `EXTENDSIZE` and `TRUNCATE_ON_CLOSE` keywords.

KEYED

Set this keyword to indicate that the file has indexed organization. Indexed files are discussed in “[VMS-Specific Information](#)” in Chapter 8 of *Building IDL Applications*.

LIST

Set this keyword to specify carriage-return carriage control when creating a new file. If no carriage-control keyword is specified, `LIST` is the default.

NONE

Set this keyword to specify explicit carriage control when creating a new file. When using explicit carriage control, VMS does not add any carriage control information to the file, and the user must explicitly add any desired carriage control to the data being written to the file.

PRINT

Set this keyword to send the file to SYS\$PRINT, the default system printer, when it is closed.

SEGMENTED

Set this keyword to indicate that the file has VMS FORTRAN-style segmented records. Segmented records are a method by which FORTRAN allows logical records to exist with record sizes that exceed the maximum possible physical record sizes supported by VMS. Segmented record files are useful primarily for passing data between FORTRAN and IDL programs.

SHARED

Set this keyword to allow other processes read and write access to the file in parallel with IDL. If SHARED is not set, read-only files are opened for read sharing and read/write files are not shared. The SHARED keyword cannot be used with STREAM files.

Warning

It is not a good idea to allow shared write access to files open in RMS block mode. In block mode, VMS cannot perform the usual record locking that prevents file corruption. It is therefore possible for multiple writers to corrupt a block mode file. This warning also applies to fixed-length record disk files, which are also processed in block mode. When using SHARED, do not specify either BLOCK or UDF_BLOCK.

STREAM

Set this keyword to open the file in stream mode using the Standard C Library (stdio).

SUBMIT

Set this keyword to submit the file to SYS\$BATCH, the default system batch queue, when it is closed.

SUPERSEDE

Set this keyword to allow an existing file to be superseded by a new file of the same name, type, and version.

TRUNCATE_ON_CLOSE

Set this keyword to free any unused disk space allocated to the file when the file is closed. This keyword can be used to get rid of excess allocations caused by the `EXTENDSIZE` and `INITIALSIZE` keywords. If the `SHARED` keyword is set, or the file is open for read-only access, `TRUNCATE_ON_CLOSE` has no effect.

UDF_BLOCK

Set this keyword to create a file similar to those created with the `BLOCK` keyword except that new files are created with the RMS undefined record type. Files created in this way can only be accessed by IDL in block mode, and cannot be processed by many VMS utilities. Do not specify both `UDF_BLOCK` and `SHARED`.

VARIABLE

Set this keyword to indicate that the file has variable-length records. If the *Record_Length* argument is present, it specifies the maximum record size. Otherwise, the only limit is that imposed by RMS (32767 bytes). If no file organization is specified, variable-length records are the default.

Warning

VMS variable length records have a 2-byte record-length descriptor at the beginning of each record. Because the `FSTAT` function returns the length of the data file *including* the record descriptors, reading a file with VMS variable length records into a byte array of the size returned by `FSTAT` will result in an RMS EOF error.

Windows-Only Keywords

The Windows-Only keywords `BINARY` and `NOAUTOMODE` are now obsolete. Input/Output on Windows is now handled identically to Unix, and does not require you to be concerned about the difference between “text” and “binary” modes. These keywords are still accepted for backwards compatibility, but are ignored.

BINARY

Set this keyword to treat opened files as binary files. When writing text to a binary file, CR/LF pairs are written as LF only. Note that setting the `BINARY` keyword alone does not ensure that a routine that writes to the file will not change the mode to text.

NOAUTOMODE

Set this keyword to prevent IDL routines such as **PRINTF** from automatically changing the mode from binary to text, or vice versa.

PRINT/PRINTF

The following keywords to the two PRINT procedures are obsolete.

VMS Keywords

REWRITE

When writing data to a file with indexed organization, set the REWRITE keyword to specify that the data should update the contents of the most recently input record instead of creating a new record.

READ_TIFF

The following keywords to the READ_TIFF function are obsolete.

Keywords

ORDER

Set this keyword to a named variable that will contain the order value from the TIFF file. This value is returned as 0 for images written bottom to top, and 1 for images written top to bottom. If an order value does not appear in the TIFF file, an order of 1 is returned.

The ORDER keyword can return any of the following additional values (depending on the source of the TIFF file):

Rows	Columns
1	top to bottom, left to right
2	top to bottom, right to left
3	bottom to top, right to left
4	bottom to top, left to right
5	top to bottom, left to right
6	top to bottom, right to left
7	bottom to top, right to left
8	bottom to top, left to right

Table 1: Values for the ORDER keyword

Reference: Aldus TIFF 6.0 spec (TIFF version 42).

UNSIGNED

This keyword is now obsolete because older versions of IDL did not support the unsigned 16-bit integer data type. Set this keyword to return TIFF files containing unsigned 16-bit integers as signed 32-bit longword arrays. If not set, return an unsigned 16-bit integer for these files. This keyword has no effect if the input file does not contain 16-bit integers.

READ/READF

The following keywords to the READ procedures are obsolete.

VMS Keywords

Note also that the obsolete VMS-only routine READ_KEY has been replaced by the keywords below.

KEY_ID

The index key to be used (primary = 0, first alternate key = 1, etc...) when accessing data from a file with indexed organization. If this keyword is omitted, the primary key is used.

KEY_MATCH

The relation to be used when matching the supplied key with key field values (EQ = 0, GE = 1, GT = 2) when accessing data from a file with indexed organization. If this keyword is omitted, the equality relation (0) is used.

KEY_VALUE

The value of a key to be found when accessing data from a file with indexed organization. This value must match the key definition that is determined when the file was created in terms of type and size—no conversions are performed. If this keyword is omitted, the next sequential record is used.

READU

The following keywords to the READU procedure are obsolete.

VMS-Only Keywords

Note

The obsolete VMS routines FORRD, and FORRD_KEY have been replaced by the READU command used with the following keywords.

KEY_ID

The index key to be used (primary = 0, first alternate key = 1, etc...) when accessing data from a file with indexed organization. If this keyword is omitted, the primary key is used.

KEY_MATCH

The relation to be used when matching the supplied key with key field values (EQ = 0, GE = 1, GT = 2) when accessing data from a file with indexed organization. If this keyword is omitted, the equality relation (0) is used.

KEY_VALUE

The value of a key to be found when accessing data from a file with indexed organization. This value must match the key definition that is determined when the file was created in terms of type and size—no conversions are performed. If this keyword is omitted, the previous key value is used.

SAVE

The following keywords to the SAVE procedure are obsolete.

Keywords

XDR

This keyword is obsolete and will be quietly ignored (there is no need to remove uses of the XDR keyword from existing code). IDL always generates XDR format files, although it will continue to read VAX format SAVE files generated by old versions of VMS IDL.

SPAWN

The following keywords to the SPAWN procedure are obsolete.

Keywords

FORCE

Set this keyword to override buffered file output in IDL and force the file to be closed no matter what errors occur in the process. If it is not possible to properly flush this data when a file close is requested, an error is normally issued and the file remains open. An example of this might be that your disk does not have room to write the remaining data. This default behavior prevents data from being lost, but the FORCE keyword overrides this behavior.

Macintosh-Only Keywords

MACCREATOR

Use this keyword to specify a four-character scalar string containing the Macintosh file creator code of the application to be used to open the specified files. In no files were specified, the application is launched without any files.

VMS-Only Keywords

NOCLISYM

If this keyword is set, the spawned subprocess does not inherit command language interpreter symbols from its parent process. You can specify this keyword to prevent commands redefined by symbol assignments from affecting the spawned commands, or to speed process startup.

NOLOGNAM

If this keyword is set, the spawned subprocess does not inherit process logical names from its parent process. You can specify this keyword to prevent commands redefined by logical name assignments from affecting the spawned commands, or to speed process startup.

NOTIFY

If this keyword is set, a message is broadcast to `SYSS$OUTPUT` when the child process completes or aborts. `NOTIFY` has no effect unless `NOWAIT` is set.

WIDGET_BASE

The following keywords to the WIDGET_BASE function are obsolete.

Keywords

APP_MBAR

Set this keyword to a named variable that defines a widget application's menubar. On the Macintosh, the menubar defined by APP_MBAR becomes the system menubar (the menubar at the top of the Macintosh screen). On Motif platforms and under Microsoft Windows, the APP_MBAR is treated in exactly the same fashion as the menubar created with the MBAR keyword. See “[MBAR](#)” on page 2115 for details on creating menubars.

Warning

You cannot specify both an APP_MBAR and an MBAR for the same top-level base widget. Doing so will cause an error.

To apply actions triggered by menu items to widgets other than the base that includes the menubar, use the [KBRD_FOCUS_EVENTS](#) keyword to keep track of which widget has (or last had) the keyboard focus.

WRITE_TIFF

The following features of the WRITE_TIFF procedure are obsolete.

Arguments

ORDER

This argument should be 0 if the image is stored from bottom to top (the default). For images stored from top to bottom, this argument should be 1.

Warning

Not all TIFF readers honor the value of the Order argument. IDL writes the value into the file, but many known readers ignore this value. In such cases, we recommend that you convert the image to top to bottom order with the REVERSE function and then set Order to 1.

WRITEU

The following keywords to the WRITEU procedure are obsolete.

VMS-Only Keywords

Note

The obsolete FORWRT routine has been replaced by WRITEU.

REWRITE

When writing data to a file with indexed organization, setting the REWRITE keyword specifies that the data should update the contents of the most recently input record instead of creating a new record.



Chapter 4: Obsolete Graphics Devices

This chapter contains documentation for graphics devices that are no longer supported by IDL. If you attempt to set IDL's graphics device to be one of the devices listed in this chapter via the `SET_PLOT` procedure, IDL will generate an error like

```
% Graphics device not available: device
```

For information on keywords to the `DEVICE` procedure that have become obsolete along with these graphics devices, see the `DEVICE` section of [Chapter 3, “Routines with Obsolete Keywords”](#).

The LJ Device

Device Keywords Accepted by the LJ Device:

CLOSE_FILE, DEPTH, FILENAME, FLOYD, INCHES, LANDSCAPE, ORDERED, PIXELS, PORTRAIT, RESOLUTION, SET_CHARACTER_SIZE, THRESHOLD, XOFFSET, XSIZE, YOFFSET, YSIZE.

The LJ250 and LJ252 are color printers sold by Digital Equipment Corporation (DEC). To direct graphics output to a picture description file compatible with these printers, issue the command:

```
SET_PLOT, 'LJ'
```

This causes IDL to use the LJ driver for producing graphical output. To actually print the generated graphics, send the file to the printer using the normal printing facilities supplied by the operating system. Once the LJ driver is enabled via SET_PLOT, the DEVICE procedure is used to control its actions, as described below. The default settings for the LJ driver are given in the following table. Use the `HELP, /DEVICE` command to view the current font, file, and other options currently set for LJ output.

Feature	Value
File	idl.lj
Mode	Portrait
Dither method	Floyd-Steinberg
Resolution	180 dpi
Number of planes	1 (monochrome)
Horizontal offset	1/2 in.
Vertical offset	1 in.
Width	7 in.
Height	5 in.

Table 4-1: Default LJ Driver Settings

LJ Driver Strengths

The LJ250 produces color graphics at a low cost. It is capable of producing good quality monochrome output, and is also good at color vector graphics and simple color imaging using a small number of predefined solid colors.

LJ Driver Limitations

The LJ250 is intended to be used as a low cost printer for business color graphics. Although it can be used to print color images, it is limited in its ability to produce satisfactory images of the sort commonly encountered in science and engineering. These limitations make it a poor choice for such work.

- Although color is specified via the usual RGB triples using the TVLCT procedure, the LJ250 is only capable of generating a fixed set of colors. The number of possible colors depends on the resolution in use. When producing 180 dpi graphics, only the colors given in the following table are possible. In 90 dpi mode, 256 colors are available.

Color	Red Value	Green Value	Blue Value
Black	10	10	10
Yellow	227	212	33
Magenta	135	13	64
Cyan	5	56	163
Red	135	20	36
Green	8	66	56
Blue	10	10	74
White	229	224	217

Table 4-2: LJ250 Colors Available at 180 dpi

If a color is specified that the printer cannot produce, it substitutes the closest color it can. However, the results of such substitutions can give unexpected results. The fixed set of possible colors means that the LOADCT procedure is of limited use with the LJ250. It also means that it is difficult to produce satisfactory grayscale images.

- The number of simultaneous colors possible on an output page is limited. Although images are specified in 8-bit bytes, the number of significant bits used ranges from 1 to 4 (as specified via the DEPTH keyword to the DEVICE procedure), allowing from 2 to 16 colors. Coupled with the above limitation on the colors that are possible, it is difficult to produce high quality image output.

LJ Suggestions

The following suggestions are intended to help you get the most out of the LJ250, taking its limitations into account:

- Use monochrome output when possible. This results in considerably smaller output files, and provides most of the abilities the LJ250 handles well. When producing monochrome output, the LJ250 driver dithers images. This can often produce more satisfying grayscale output than is possible using the printer in color mode.
- The table under “[LJ Driver Limitations](#)” above gives the RGB values to use when specifying colors at 180 dpi. To make more colors available, use 90 dpi resolution. The RGB values for the possible colors at 90 dpi are given in Table 7-6 of the *LJ250/LJ252 Companion Color Printer Programmer Reference Manual*. You can cause the printer to display the complete 256 color palette as follows: With the power off, press and hold the READY and DEC/PCL switches while momentarily pressing the power switch. Wait approximately 2 seconds and release the READY and DEC/PCL switches. The printer will take a few minutes to print all 256 colors. The output fits on a single page.

Use the table in the programmers manual with this display to select the colors to use. Note that the RGB values in the programmers manual are scaled from 1 to 100, while IDL scales such values from 0 to 255. Therefore, multiply the values obtained from the manual by 2.55 to properly scale them for use in IDL.

- Unlike most devices, IDL does not initialize the LJ250 color map to a grayscale ramp because the printer cannot produce a satisfactory grayscale image. Instead, the default palettes given in Table 7-5 of the *LJ250/LJ252 Companion Color Printer Programmer Reference Manual* are used. If you modify the color map, the LJLCT procedure can be used to reset the color table to these defaults. LJLCT examines the !D.N_COLORS system variable to determine the number of output planes in use, then loads the appropriate default color map.
- When producing images, stick to images with small amounts of detail and large sections of uniform color. Complicated images do not reproduce well on this printer.

The Macintosh Device

Device Keywords Accepted by the MAC Device:

BYPASS_TRANSLATION, COPY, CURSOR_ORIGINAL,
CURSOR_STANDARD, DECOMPOSED, FLOYD, GET_CURRENT_FONT,
GET_FONTNAMES, GET_FONTNUM, GET_GRAPHICS_FUNCTION,
GET_SCREEN_SIZE, GET_WINDOW_POSITION, ORDERED,
PSEUDO_COLOR, RETAIN, SET_CHARACTER_SIZE, SET_FONT,
SET_GRAPHICS_FUNCTION, THRESHOLD, TRANSLATION, TRUE_COLOR

The Macintosh version of IDL uses the “MAC” device by default. This device is similar to [The X Windows Device](#). The “MAC” device is only available in IDL for Macintosh.

To set plotting to the Macintosh device, use the command:

```
SET_PLOT, 'MAC'
```




Chapter 5: Remote Procedure Calls

Note

Remote Procedure Calls are still included in IDL. The RPC API described here (the API included with IDL version 4.0) has been replaced with a new API. See the External Development Guide for details on the RPC API included with IDL version 5.0 and later.

Remote Procedure Calls (RPCs) allow one process (the *client* process) to have another process (the *server* process) execute a procedure call just as if the caller process had executed the procedure call in its own address space. Since the client and server are separate processes, they can reside on the same machine or on different machines. RPC libraries allow the creation of network applications without having to worry about underlying networking mechanisms.

IDL supports RPCs so that other applications can communicate with IDL. A library of C language routines is included to handle communication between client programs and the IDL server. *Note that remote procedure calls are supported only on UNIX platforms.*

The current implementation allows IDL to be run as an RPC server and your own program to be run as a client. IDL commands can be sent from your application to the IDL server, where they are executed. Variable structures can be defined in the client program and then sent to the IDL server for creation as IDL variables. Similarly, the values of variables in the IDL server session can be retrieved into the client process.

Using IDL as an RPC Server

The IDL RPC Directory

All of the files related to using IDL's RPC capabilities are found in the `rpc` subdirectory of the `external` subdirectory of the main IDL directory. The main IDL directory is referred to here as *idldir*.

Running IDL in Server Mode

To use IDL as an RPC server, run IDL in server mode by using the `-server` command line option. This option can be invoked one of two ways:

```
idl -server process_id
```

or

```
idl -server=server_number process_id
```

where *server_number* is the hexadecimal server ID number (between 0x20000000 and 0x3FFFFFFF) for IDL to use. For example, to run IDL with the server ID number 0x20500000, use the command:

```
idl -server=20500000
```

If a server ID number is not supplied, IDL uses the default, `IDL_DEFAULT_ID`, defined in the file *idldir/external/rpc/rpc_idl.h*. This value is originally set to 0x2010CAFE.

The *process_id* argument is an optional argument that specifies the process ID of a UNIX process that should be contacted when IDL has finished running in interactive mode. If the IDL rpc server is placed in interactive mode and a process ID has been supplied on the command line, IDL sends the UNIX signal `SIGUSR1` to the specified process. This signal allows the client program to know when it can continue to communicate with the rpc server.

Creating the IDL RPC Library

The machine that runs the client program must have its own version of the IDL RPC library. The make file for this library is contained in the directory *idldir/external/rpc*. If the machine that runs the client program is not licensed to run IDL, simply copy the contents of the IDL `rpc` directory to an appropriate location on the client machine.

To build the IDL RPC library, copy the IDL `rpc` directory to a new directory, change to that directory, and enter the make command:

```
cp -R idldir/external/rpc newrpcdir
cd newrpcdir
make
```

The created library is contained in the file `newrpcdir/rpcidl.a`. The functions contained in the library are described in [“The IDL RPC Library”](#) on page 133

Linking your Client Program

Your client program must include the file `idldir/external/rpc/rpc_idl.h`.

You must also link the application that communicates with IDL with the IDL RPC library. For example, to compile and link a program with the IDL RPC library, you might enter:

```
cc -c rpcclient.c
cc -o rpcclient.o idldir/external/rpc/rpcidl.a
```

where `rpcclient.c` is the name of your program. Note that your actual command lines and flag settings may be different than the ones shown above, depending upon your C compiler. The `Makefile` contains details on modifications for various systems.

The IDL RPC Library

The IDL RPC library contains several C language interface functions that facilitate communication between your application and IDL. There are functions to register and unregister clients, set timeouts, get and set the value of IDL variables, send commands to the IDL server, and cause the server to exit. These functions are described below.

free_idl_variable

Syntax

```
void free_idl_var(varinfo_t* var);
```

Description

This function frees all dynamic memory associated with the given variable. Attempts to free a static variable are silently ignored. (See [“Notes on Variable Creation and Memory Management”](#) on page 158)

Parameters

var

The address of the varinfo_t structure that contains the information about the variable to be freed.

Return Value

None

get_idl_variable

Syntax

```
int get_idl_variable(CLIENT* client, char* name, varinfo_t* var,
                    int typecode)
```

Description

Call this function to retrieve the value of an IDL variable in the IDL session referred to by client. Any scalar or array variable type can be retrieved. Variables can be retrieved only from the main program level.

Note that it is not possible to get the value of an IDL structure. To retrieve values from an IDL structure, “decompose” the structure into regular variables in IDL, then use this function to get the values of those individual variables.

It is not possible to get the value of IDL system variables directly. To retrieve the value of an IDL system variable, first copy it to a regular IDL variable. The value of the regular variable can then be retrieved with `get_idl_variable`. For example:

```
varinfo_t pt; /* Declare variable pt */
send_idl_command(client, "X = !P.T");
get_idl_variable(client, "X", &pt, 0);
```

Parameters

client

A pointer to the CLIENT structure that corresponds to the desired IDL session.

name

A null terminated string that contains the name of the IDL variable to be retrieved. Only the first MAXIDLEN characters of this string are used. MAXIDLEN is defined in the file `idldir/external/rpc/rpc_idl.h`.

var

The address of a `varinfo_t` structure in which to store the returned variable information. Upon return, the Name field of the var structure contains the name of the variable as found in IDL. If the name supplied is an illegal IDL variable name, the Name field is set to `<ILLEGAL_NAME>`. If the variable is a structure or associated variable, the Name field is set to `<BAD-VAR-TYPE>`.

typecode

If you want IDL to typecast a variable (i.e., guarantee the value to be of a particular type) before it is transported, set `typecode` to one of the following values (defined in the file `export.h`):

```
IDL_TYP_BYTE, IDL_TYP_INT, IDL_TYP_LONG, IDL_TYP_FLOAT,  
IDL_TYP_DOUBLE, IDL_TYP_STRING, IDL_TYP_COMPLEX, IDL_TYP_DCOMPLEX
```

For example, the command:

```
get_idl_variable(client, "x", &xv, IDL_TYP_LONG)
```

guarantees that the value in `x` is returned as a 32-bit integer.

If `typecode` is 0, the variable is transferred with whatever data type it has in the server. Typecasting only affects the variables in the client – the server side is not affected.

Return Value

This function returns a status value that denotes the success or failure of this function as described below.

- 1 Failure: bad arguments supplied (e.g., name or var is NULL).
- 0 RPC mechanism failed (an error message may also be printed).
- 1 Success
- 2 Illegal variable name (e.g., “213xyz”, “#a”, “!DEVICE”)
- 3 Variable not transportable (e.g., the variable is a structure or associated variable)

idl_server_interactive

Syntax

```
int idl_server_interactive(CLIENT*client)
```

Description

Call this function to cause the IDL server to become an interactive IDL session. It is likely that this command will time out. Some alternative mechanism for determining when the server is finished should be implemented. See the example `server.c` in the `idldir/examples/rpc` directory.

Parameters

client

A CLIENT structure that corresponds to the desired IDL session.

Return Value

This function returns TRUE if the interactive IDL session did not time out. FALSE is returned if the session times out or otherwise fails.

kill_server

Syntax

```
int kill_server(CLIENT*client)
```

Description

Call this function to kill the IDL RPC server.

Parameters

client

The pointer to a CLIENT structure registered with the server to be killed.

Return Value

This function returns TRUE if the server was successfully killed. FALSE is returned otherwise.

register_idl_client

Syntax

```
CLIENT* register_idl_client(long server_id, char* hostname,  
                             struct timeval* timeout)
```

Description

Call this function to register your program as a client of an IDL server. Note that a program can be the client of a number of different servers at the same time and a single server can have multiple clients.

Parameters

server_id

The ID number of the IDL server that the program is to be registered with. If this value is 0, the default server ID (0x2010CAFE) is used.

hostname

The name of the machine where the IDL server is running. If this value is NULL or " ", the default, localhost, is used.

timeout

A pointer to the timeout value for all communication with IDL servers. If this value is NULL or 0, the default timeout, 60 seconds, is used.

Return Value

A pointer to the new CLIENT structure is returned. This function returns NULL if it is unsuccessful.

send_idl_command

Syntax

```
int send_idl_command(CLIENT* client, char* command);
```

Description

Call this function to send an IDL command to the IDL server referred to by client. The command is executed just as if it had been entered from the IDL command line.

This function cannot be used to send multi-line commands. If the first part of a multi-line command is sent, for example:

```
send_idl_command(client, "FOR I=1,5 DO $");
```

IDL spawns an interactive session and may hang. In any case, subsequent commands are not executed.

Parameters

client

A pointer to the CLIENT structure that corresponds to the desired IDL session.

command

A null-terminated string with no more than MAX_STRING_LEN characters. MAX_STRING_LEN is defined in the file *idldir/external/rpc/rpc_idl.h*.

Return Value

This function returns a status value that denotes success or failure as described below.

- -1 = RPC communication failure (an error message is also printed).
- 0 = Command is NULL.
- 1 = Success.

For all other errors, the error number is returned. This number could be passed as an argument to STRMESSAGE () ; .

set_idl_timeout

Syntax

```
int set_idl_timeout(struct timeval* timeout)
```

Description

Call this function to replace the current timeout used by the RPC mechanism with the given timeout.

Parameters

timeout

A pointer to the new timeout value to be used. This parameter has no default.

Return Value

This function returns TRUE if the timeout was replaced. FALSE is returned if the timeout value was NULL or zero.

set_idl_variable

Syntax

```
int set_idl_variable(CLIENT* client, varinfo_t* var);
```

Description

Call this function to assign a value to an IDL variable in the IDL session referred to by `client`. The address `var` points to a `varinfo_t` structure that contains information about the variable to be set. The “helper” functions can be used to build `var`. (See “[The varinfo_t Structure](#)” on page 146) Any scalar or array variable type can be set. Variables can be set only in the main IDL program level.

Note that it is not possible to set the value of an IDL structure. To set values in an IDL structure, set the individual elements of the structure to scalar IDL variables, then use the `send_idl_command` function to create the structure in IDL.

It is not possible to set the value of IDL system variables directly. To set the value of an IDL system variable, first set the value of a regular IDL variable. The value of the regular variable can then be assigned to the system variable. For example:

```
set_idl_variable(client, &newvar); /* newvar describes the */
                                   /* IDL variable "NEW" */
send_idl_command(client, "!P.T = NEW");
```

Parameters

client

A pointer to the `CLIENT` structure that corresponds to the desired IDL session.

var

The address of the `varinfo_t` structure that contains information about the variable to be set.

Return Value

This function returns a status value that denotes the success or failure of this function as described below.

- -1 = Failure: bad arguments supplied (e.g., `var` is `NULL`).

- 0 = RPC mechanism failed (an error message is also printed).
- 1 = Success

set_rpc_verbosity

Syntax

```
void set_rpc_verbosity(verbosity)
```

Description

This function controls the printing of error messages by RPC library routines. If `verbosity` is `TRUE`, error messages will be printed by the various RPC routines to explain what failed. If `verbosity` is `FALSE`, return codes continue to indicate success or failure, but no error messages are printed.

Parameters

verbosity

An `int` specifying `TRUE` or `FALSE` as explained above.

Return Value

None

unregister_idl_client

Syntax

```
void unregister_idl_client(CLIENT* client)
```

Description

Call this function to release the resources associated with the given CLIENT structure. The operating system automatically releases the resources associated with all CLIENT structures when your program exits. This function does not affect the IDL server.

Parameters

client

The pointer to the CLIENT structure to be unregistered.

Return Value

None

The varinfo_t Structure

The `varinfo_t` structure is used to pass variables to and from the IDL server.

The `varinfo_t` structure is defined in the `idldir/external/rpc/rpc_idl.h` file. The structure is:

```
typedef struct _VARINFO {
    char Name[MAXIDLEN+1];
    IDL_VPTR Variable;
    IDL_LONG Length;
} varinfo_t;
```

Variable Creation Functions

A number of functions are provided to help build `varinfo_t` structures. These functions are contained in the file `idldir/external/rpc/helper.c`.

The variable creation functions are described below. Unless otherwise noted, all of the following functions return `TRUE` if variable creation is successful and `FALSE` otherwise. When passing a `varinfo_t` structure pointer, if the `Variable` field is `NULL`, the variable creation functions attempt to allocate that field.

v_make_byte

Syntax

```
int v_make_byte(varinfo_t* var_struct, char* var_name,  
               unsigned value)
```

Description

Create an IDL byte variable with the given name and value.

v_make_complex

Syntax

```
int v_make_complex(varinfo_t* var_struct, char* var_name,  
                  double real_value, double imag_value)
```

Description

Create an IDL complex variable.

v_make_dcomplex

Syntax

```
int v_make_dcomplex(varinfo_t* var_struct, char* var_name,  
                    double real_value, double imag_value)
```

Description

Create an IDL double-precision complex variable.

v_make_double

Syntax

```
int v_make_double(varinfo_t* var_struct, char* var_name,  
                 double value)
```

Description

Create an IDL double-precision, floating-point variable.

v_make_float

Syntax

```
int v_make_float(varinfo_t* var_struct, char* var_name,  
                double value)
```

Description

Create an IDL single-precision, floating-point variable.

v_make_int

Syntax

```
int v_make_int(varinfo_t* var_struct, char* var_name, int value)
```

Description

Create an IDL (16-bit) integer variable.

v_make_long

Syntax

```
int v_make_long(varinfo_t* var_struct, char* var_name,  
                IDL_LONG value)
```

Description

Create an IDL long variable.

v_make_string

Syntax

```
int v_make_string(varinfo_t* var_struct, char* name,  
                 char* value)
```

Description

Create an IDL string variable.

v_fill_array

Syntax

```
int v_fill_array(varinfo_t* var, char* name, int type,
                int ndimension, IDL_LONG dims[], UCHAR* value,
                IDL_long length)
```

Description

Create an IDL array variable. The *type* argument should be one of the following values (defined in the file `export.h`):

```
IDL_TYP_BYTE, IDL_TYP_INT, IDL_TYP_LONG, IDL_TYP_FLOAT,
IDL_TYP_DOUBLE, IDL_TYP_STRING, IDL_TYP_COMPLEX, IDL_TYP_DCOMPLEX
```

This function allocates `var->Variable->value.arr`.

If `value` is `NULL` then `var->Variable->value.arr->data` is allocated.

The `dims[]` argument should have at least `ndimension` valid elements.

If `value` is supplied but `length` is 0, `var->Length` is filled with the computed size of the array (in bytes) and `value` is assumed to point to at least that many bytes of memory. If `value` and `length` are supplied, `length` is assumed to be the size (in bytes) of the region of memory that `value` points to. (See [“Notes on Variable Creation and Memory Management”](#) on page 158)

More Variable Manipulation Macros

The following macros can be used to get information from `varinfo_t` structures. Like the variable creation functions, these macros are defined in the file `rpc_idl.h`.

All of these macros accept a single argument `v` of `varinfo_t` type.

GetArrayData(*v*)

This macro returns a pointer to the array data described by the `varinfo_t` structure.

GetArrayDimensions(*v*)

This macro returns the dimensions of the array described by the `varinfo_t` structure. The dimensions are returned as `long dimensions[]`.

GetArrayNumDims(*v*)

This macro returns the number of dimensions of the array.

GetVarByte(*v*)

This macro returns the value of a 1-byte, unsigned `char` variable.

GetVarComplex(*v*)

This macro returns the value (as a *struct*, not a pointer) of a complex variable.

GetVarDComplex(*v*)

This macro returns the value (as a *struct*, not a pointer) of a double-precision, complex variable.

GetVarDouble(*v*)

This macro returns the value of a double-precision, floating-point variable.

GetVarFloat(*v*)

This macro returns the value of a single-precision, floating point variable.

GetVarInt(*v*)

This macro returns the value of a 2-byte integer variable.

GetVarLong(*v*)

This macro returns the value of a 4-byte integer variable.

GetVarString(*v*)

This macro returns the value of a string variable (as a *char**).

GetVarType(*v*)

This macro returns the type of the variable described by the `varinfo_t` structure. The type is returned as `IDL_TYP_XXX` as described under the documentation for the `get_idl_variable` function.

VarIsArray(*v*)

This macro returns non-zero if *v* is an array variable.

Notes on Variable Creation and Memory Management

This section contains miscellaneous notes about variable creation.

Freeing Resources

The variable creation functions (i.e., `v_make_xxx`) *do not* free resources associated with a variable before placing new information there. Your programs should free resources (if there are any) associated with the `varinfo_t` structure being passed.

To prevent memory leakage, memory associated with a variable is freed before new memory is allocated. You should make sure that the `varinfo_t` structure passed to the `get_idl_variable` function contains valid information or has been cleared (to zeroes) first. If an array of the same size, dimensions, and type is being read into the existing array variable, no allocation is performed and the same space is re-used. For example:

```
/* Assume that:
   X = FLTARR(1000, 1000)
   Y = FLTARR(1000, 1000)
   Z = LONARR(1000, 1000) same size, different type
*/
bzero(&vinfo, sizeof(vinfo));
get_idl_variable(client, "X", &vinfo, 0); /* array allocated */
...
get_idl_variable(client, "Y", &vinfo, 0); /* memory re-used */
...
get_idl_variable(client, "Z", &vinfo, 0); /* array allocated */
free_idl_var(&vinfo);
```

The `get_idl_variable` function calls `free_idl_var` before doing any allocation. So, in the example above, we only needed to free Z. X and Y were freed when we re-used `vinfo`.

Creating a Statically-Allocated Array

It is possible to create a statically-allocated array for receiving information from the server without having the overhead of memory reallocation every time information is received.

If the `Length` field of the `varinfo_t` structure is not zero, it is assumed to be the size of the array data. The `free_idl_var` function will not do anything to a variable where `length` is non-zero. It is up to the programmer to do their own memory

management if this is the case. Storing a scalar in a static variable (i.e., a variable that has a non-zero `Length` field) fails as does attempting to store an array that does not fit the statically-allocated array. For example:

```

/* X = FLTARR(10)      40 bytes of data (10*4)
   Y = LONARR(2,2,2)   32 bytes of data(2*2*2*4)
   Z = BYTARR(50)      50 bytes of data
   W = 12               scalar
*/
char      buf[40]
varinfo_t v;
VARIABLE  var;
ARRAY     arr;
/* Build a static array. Fill in the minimum amount of */
/* information required.                                */
v.Variable = &var;
v.Length   = 40;
var.type    = IDL_TYP_BYTE;
var.flags   = V_ARR;
var.value.arr = &arr;
arr.data    = buf;
get_idl_variable(client, "X", &v, 0); /* ok */
get_idl_variable(client, "Y", &v, 0); /* ok */
get_idl_variable(client, "Z", &v, 0); /* fails - too big */
get_idl_variable(client, "W", &v, 0); /* fails - scalar */

```

Allocating Space for Strings

All space for strings is assumed to be obtained via `malloc(3)`. This fact is important only when receiving variables (using the `get_idl_variable` function). For example, the following code fragment is valid:

```

v_make_string(&foo, "UGH", "blug");
set_idl_variable(client, &foo);

```

Here is an example of code that will crash your program:

```

v_make_string(&foo, "UGH", "blug");
set_idl_variable(me, &foo);
send_idl_command(me, "UGH='hello world'");
get_idl_variable(me, "UGH", &foo, 0);

```

In this case, the `get_idl_variable` function attempts to free the old resources before allocating new storage. Freeing the constant `blug` results in an error. You could achieve the desired result without an error by changing the first line to:

```

v_make_string(&foo, "UGH", strdup("blug"));

```

RPC Examples

A number of example files are included in the *idldir/external/examples/rpc* directory. A *Makefile* for these examples is also included. These short C programs demonstrate the use of the IDL RPC library.